

# Java Bytecode-Generierung im Rahmen eines Softwarepraktikums in HASKELL

Volker Stolz

`stolz@i2.informatik.rwth-aachen.de`

LEHRSTUHL FÜR INFORMATIK II

**RWTH** Aachen



# Überblick

---

- Motivation: Haskell in der Lehre
- Praktikum: Compilerbau
- Java Virtual Machine & Bytecode
- Der Java Bytecode Verifier
- Codegenerierung
- Ausgabe in Java
- Zusammenfassung



# Softwarepraktikum in HASKELL

Im Softwarepraktikum wird den Studenten Wissen auf drei Teilgebieten vermittelt:

- Programmieren in Haskell
  - Funktionale Programmierung
  - Implementierung einer Stack-Maschine mit Hilfe algebraischer Datentypen für die einzelnen „Assembler“-Befehle
  - Zustandsmonade für Einzelschrittsemantik bei Programmausführung  
(PC erhöhen, lokale Variablen modifizieren...)
- Compilerbau
- Im Team ein kleineres Projekt gemeinsam bearbeiten



# Zu übersetzende Programmiersprache: PSA

## PSA:

- PASCAL-ähnliche Syntax
- `if` und `while` als Kontrollstrukturen
- Keine Prozeduren, einziger Datentyp Integer
- Variablen müssen vor ihrer Verwendung deklariert werden

## Ein/Ausgabe:

- Parameter an das Programm werden vor dem Start in lokalen Variablen angegeben
- Maschine liefert am Ende des Programms einen Dump der Variableninhalte
- Explizite Ein/Ausgabe erst in letzter Praktikumsaufgabe



# Die Praktikumsaufgabe

Sieben Teilaufgaben müssen in jeweils zwei Wochen in Zweiergruppen bearbeitet werden:

- Scanner (pure funktionale Programmierung)
- Parser (monadische Parserkombinatoren)
- Semantische Analyse (Tabelle als Funktion)
- Abstrakte Maschine (Zustandsmonade)
- Codegenerierung für die abstrakte Maschine (*laziness*)
- Web-Frontend (HTML-Kombinatoren)
- Erweiterung um Ein- und Ausgabe



# Beispielprogramm: GGT (PSA)

```
var x y;  
x := 42  
y := 15  
while x /= y do  
  if x > y then  
    x := (x - y)  
  else  
    y := (y - x)  
  end  
end.  
end.
```



# Bestandteile der A-Maschine

## Zustand

1. Befehlszähler
2. Stack
3. Hauptspeicher mit lokalen Variablen

## Befehlssatz

- `Lit z`
- `Sto n`
- `Lod n`
- `Jmp a / Jmc a`
- `Ad, Sm, Ml, Dv` (arithmetische Operationen)
- `Eq, Ne, Gt, Ge, Lt, Le` (Relationen)
- `Halt`



# Beispielprogramm: GGT (PSX)

```
0:  Lit 42
    Sto 1
    Lit 15
    Sto 2
4:  Lod 1 ; while
    Lod 2
    Ne
    Jmc 22 ; Sprung hinter while
    Lod 1
    Lod 2
    Gt
                                Jmc 17
                                Lod 1
                                Lod 2
                                Sb
                                Sto 1
                                Jmp 21
17: Lod 2
    Lod 1
    Sb
    Sto 2
21: Jmp 4 ; zurueck zu while
22: Halt
```



# Codegenerator für Java Bytecode

## Beobachtung:

Die JAVA VIRTUAL MACHINE (JVM) entspricht (fast) unserer Stackmaschine:

- Arithmetische Operationen erwarten Argumente auf Stack
- Lokale Variablen (insbesondere keine Register!)
- Ähnliche bedingte Verzweigungen
- [Klassen/Methoden spielen für unsere Zwecke keine Rolle]



# Codegenerator für Java Bytecode

## Beobachtung:

Die JAVA VIRTUAL MACHINE (JVM) entspricht (fast) unserer Stackmaschine:

- Arithmetische Operationen erwarten Argumente auf Stack
- Lokale Variablen (insbesondere keine Register!)
- Ähnliche bedingte Verzweigungen
- [Klassen/Methoden spielen für unsere Zwecke keine Rolle]

## Unterschied:

Keine booleschen Werte auf Stack, statt dessen bedingte Verzweigung nach arithmetischem Vergleich



# Die wichtigsten JVM-Befehle

<code>sipush c</code>	Konstante auf Stack legen
<code>istore n</code>	Integer von Stack in lokale Variable $n$
<code>iload n</code>	Integer aus lokaler Variablen $n$ auf den Stack legen
<code>isub/iadd/...</code>	Arithmetische Operationen
<code>if_icmp[eq,le,lt]</code>	Bedingte Verzweigung, wenn die beiden Integer-Werte an der Spitze des Stacks in jeweiliger Relation stehen
<code>if[eq,ne,le,lt]</code>	Verzweigung, wenn Integer auf Stack $= 0, \neq 0, \leq 0, < 0, \dots$
<code>goto offset</code>	Sprung zu Instruktion an $pc + offset$



# Java Frame

Neue Stack-Frames werden beim Methodenaufruf angelegt (hier: nur `main` wegen flacher Struktur von PSA).

Ein Frame enthält:

- Lokale Variablen (Anzahl muß zur Compilezeit bekannt sein!)
- Operanden-Stack (dt.)
- Dynamische Verweise auf den *constant pool*



# Java Bytecode Verifier

- Anzahl lokaler Variablen muß deklariert werden (max. 256 bzw. 65534)
- Maximale Stackgröße muß deklariert werden
- Theorembeweiser prüft Stackcode:
  - für jede Instruktion:
    - speichere Stack-Zustand und lokale Variablen (Typen)
  - überprüfe Typ der aktuellen Instruktion und Parameter
  - prüfe Auswirkung der Instruktion auf Stack & Variablen, prüfe nächste Instruktion mit neuem Zustand.



# Java Bytecode Verifier

- Anzahl lokaler Variablen muß deklariert werden (max. 256 bzw. 65534)
- Maximale Stackgröße muß deklariert werden
- Theorembeweiser prüft Stackcode:
  - für jede Instruktion:
    - speichere Stack-Zustand und lokale Variablen (Typen)
  - überprüfe Typ der aktuellen Instruktion und Parameter
  - prüfe Auswirkung der Instruktion auf Stack & Variablen, prüfe nächste Instruktion mit neuem Zustand.

⇒ Problem: Verzweigungen



# Bytecode-Verifikation von Verzweigungen

## Einschränkung:

Bei jedem Sprung muß der aktuelle Zustand von Stack und Variablen am Ziel einem evtl. bereits dort gespeicherten Zustand entsprechen!

## Insbesondere:

- Gleiche Anzahl von Werten auf Stack
- Typen lokaler Variablen und Werte auf dem Stack müssen *kompatibel* sein

Dabei bedeutet *kompatibel*:

1. Entweder alle Typen identisch
2. Oder Typen von Objektreferenzen können durch nächsten gemeinsamen Vorgängertyp ersetzt werden



# Beispiel Bytecode-Prüfung

→

```
sipush 4
```

```
istore 1
```

```
l1:
```

```
aconst_null
```



```
iinc 1 -1
```

```
iload 1
```

```
ifne l1
```



# Beispiel Bytecode-Prüfung

```
→      sipush 4
       istore 1
l1:
       aconst_null      4
       iinc 1 -1
       iload 1
       ifne l1
```



# Beispiel Bytecode-Prüfung

```
sipush 4
```

```
istore 1
```

```
l1:
```

→

```
aconst_null
```



```
iinc 1 -1
```

```
iload 1
```

1.Durchlauf

```
ifne l1
```



# Beispiel Bytecode-Prüfung

```
sipush 4
```

```
istore 1
```

```
l1:
```

```
aconst_null
```



*null*

→

```
iinc 1 -1
```

```
iload 1
```

1.Durchlauf

```
ifne l1
```



# Beispiel Bytecode-Prüfung

```
sipush 4
```

```
istore 1
```

```
l1:
```

```
aconst_null
```



*null*

```
iinc 1 -1
```

→

```
iload 1
```

1.Durchlauf

```
ifne l1
```



# Beispiel Bytecode-Prüfung

```
sipush 4
```

```
istore 1
```

3

```
l1:
```

```
aconst_null
```

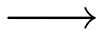
*null*

```
iinc 1 -1
```

```
iload 1
```

1.Durchlauf

```
ifne l1
```



# Beispiel Bytecode-Prüfung

```
sipush 4
```

```
istore 1
```

```
l1:
```

→

```
aconst_null
```



```
null
```

```
iinc 1 -1
```

```
iload 1
```

1.Durchlauf

Neu

```
ifne l1
```



# Bedeutung für das Praktikum?

Berührungspunkte zwischen Verifier und Praktikum:

- Lokale Variablen?
- Bytecode?
- Maximale Stackgröße?



# Bedeutung für das Praktikum?

Berührungspunkte zwischen Verifier und Praktikum:

- Lokale Variablen? Anzahl trivialerweise bekannt!
- Bytecode?
- Maximale Stackgröße?



# Bedeutung für das Praktikum?

Berührungspunkte zwischen Verifier und Praktikum:

- Lokale Variablen? Anzahl trivialerweise bekannt!
- Bytecode? Entspricht automatisch den Anforderungen des Verifiers wegen einfacher Struktur!
- Maximale Stackgröße?



# Bedeutung für das Praktikum?

Berührungspunkte zwischen Verifier und Praktikum:

- Lokale Variablen? Anzahl trivialerweise bekannt!
- Bytecode? Entspricht automatisch den Anforderungen des Verifiers wegen einfacher Struktur!
- Maximale Stackgröße? Muß während der Codegenerierung verfolgt werden!



# Bedeutung für das Praktikum?

Berührungspunkte zwischen Verifier und Praktikum:

- Lokale Variablen? Anzahl trivialerweise bekannt!
- Bytecode? Entspricht automatisch den Anforderungen des Verifiers wegen einfacher Struktur!
- Maximale Stackgröße? Muß während der Codegenerierung verfolgt werden!

Wie kann man Entwicklung des Stacks verfolgen?



# Simulation des Stacks bei Codegenerierung

## Vorgehen:

Protokolliere aktuelle und bisherige maximale Stackgröße

- `sipush/iload/istore`: klar
- Arithmetische Instruktionen: „Zwei runter, eins drauf“
- `if-then-else`: Unproblematisch, zwei Zweige
- `while bExp do stmt* end`: **Enthält Rücksprung**



# Simulation des Stacks bei Codegenerierung

## Vorgehen:

Protokolliere aktuelle und bisherige maximale Stackgröße

- `sipush/iload/istore`: klar
- Arithmetische Instruktionen: „Zwei runter, eins drauf“
- `if-then-else`: Unproblematisch, zwei Zweige
- `while bExp do stmt* end`: **Enthält Rücksprung**

## Invariante:

Nach Schleife Stack in ursprünglichem Zustand

- ↪ Stackentwicklung innerhalb der Schleife einmal simulieren, danach hinter `while`-Code fortfahren.



# Weitere Notwendigkeiten

Folgende Dinge müssen bei der Verwendung von Java noch beachtet werden:

- *Header* mit `<init>`-Methode erforderlich
- Gesamter Rumpf in `main`-Methode
- Ausgabe der lokalen Variablen notwendig



# Ausgabe mittels der JVM (1)

## Problem:

A-Maschine gibt am Ende Inhalt der lokalen Variablen aus, JVM erfordert expliziten Aufruf von `java.io.PrintStream.println`

Der erzeugte Java Bytecode benötigt *Trailer* mit dieser Funktionalität!



# Ausgabe mittels der JVM (1)

## Problem:

A-Maschine gibt am Ende Inhalt der lokalen Variablen aus, JVM erfordert expliziten Aufruf von `java.io.PrintStream.println`

Der erzeugte Java Bytecode benötigt *Trailer* mit dieser Funktionalität!

## Lösung:

Erforderlicher Code zur Ausgabe eines Wertes in lokaler Variablen:

```
getstatic java/lang/System/out Ljava/io/PrintStream;  
iload 1  
invokestatic java/lang/String/valueOf(I)Ljava/lang/String;  
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```



# Ausgabe mittels der JVM (2)

Nachteil:

Kein Iterieren über lokale Variablen von 1 bis  $n$  möglich.

Einfachste Lösung:

Obigen Code für **alle** Variablen replizieren, dabei jeweils mit `iload  $i$`  den entsprechenden Index laden.



# Ausgabe mittels der JVM (2)

Nachteil:

Kein Iterieren über lokale Variablen von 1 bis  $n$  möglich.

Einfachste Lösung:

Obigen Code für **alle** Variablen replizieren, dabei jeweils mit `iload  $i$`  den entsprechenden Index laden.



# Ausgabe mittels der JVM (2)

Nachteil:

Kein Iterieren über lokale Variablen von 1 bis  $n$  möglich.

Einfachste Lösung:

Obigen Code für **alle** Variablen replizieren, dabei jeweils mit `iload i` den entsprechenden Index laden.

- Evtl. durch Hilfsmethode etwas die Lesbarkeit erhöhen?
- Arrays benutzen?



# Revisited: GGT in Java Bytecode (1)

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

.method public <init>()V
  aload 0
  invokespecial java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit stack 2
  .limit locals 3

  <code>

end:
  getstatic java/lang/System/out Ljava/io/PrintStream;
  iload 1
  invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```



# Revisited: GGT in Java Bytecode (2)

```
    sipush 42
    istore 1
    sipush 15
    istore 2
l_4:   iload 1
       iload 2
       if_icmpeq end
       iload 1
       iload 2
       if_icmple l_17
                                l_17:  iload 2
                                iload 1
                                isub
                                istore 2
                                l_21:  goto l_4
                                iload 1
                                isub
                                istore 1
                                goto l_21
```



---

# Zusammenfassung

- Mit Haskell lassen sich auch schon im Grundstudium komplexe Praktika erfolgreich bearbeiten
- Hohe Motivation durch Einbindung von Java Bytecode
- Tieferes Wissen über die JVM
- Ausbaufähig zu einem Praktikum im Hauptstudium?
- <http://www-i2.informatik.rwth-aachen.de/Praktikum/SWP/>

## Verwendetes Werkzeug:

- Java Virtual Machine Assembler *Jasmin*  
<http://www.cat.nyu.edu/meyer/jasmin>



# Datenstrukturen der Java Laufzeitumgebung

- *program counter* PC
- Stack mit Stack-Frames
- Heap
- *method area, constant pool*
- Verwaltungsstrukturen für *native* Funktionsaufrufe (z.B. C)



# Der Java (Bytecode) Verifier

Durchgeführte Tests:

- Strukturelle Prüfung
  - Entspricht `.class`-Dateiformat
  - *constant pool* (selbst und Verweise darauf)
  - Offsets in Sprunganweisungen
  - Offsets + Typen in *constant pool*
- Umgebung: Verweise auf andere Klassen und ihre Methoden
- Inhalt (der eigentliche Bytecode)

