DIPLOMARBEIT

CACAO
EIN 64BIT - JAVAVM - JUST-IN-TIME - COMPILER

ausgeführt am

INSTITUT FÜR COMPUTERSPRACHEN
DER TECHNISCHEN UNIVERSITÄT WIEN

unter der Anleitung von

UNIV. DOZ. DIPL.-ING. DR. ANDREAS KRALL
und UNIV. ASS. DIPL.-ING. DR. ANTON ERTL
als mitwirkendem Universitätsassistenten

durch

REINHARD GRAFL
Wiesengasse 19-21/12 1090 Wien

Kurzfassung

Diese Diplomarbeit beschreibt die Implementierung von CACAO, eines Compilers, der Java-Klassenfiles in Maschinencode übersetzt.

CACAO lädt die Klassenfiles, in denen Java-Programme als Instruktionen für einen hypothetischen Prozessor - der "Java Virtual Machine" - codiert sind, in den Speicher. Anschließend übersetzt er die Programmteile, die er gerade braucht (Just In Time) in Maschinencode und führt sie aus.

Im Vergleich mit Interpretern macht sich diese Übersetzung durch 2 bis 17 mal höhere Geschwindigkeit bemerkbar. Je länger die Laufzeit der Programme im Verhältnis zu ihrer Größe ist, desto deutlicher wird diese Beschleunigung.

Ein zweiter Vorteil ist, dass bei der Übersetzung auch die Datenformate der JavaVM, insbesondere die Objekt-Zeiger, an die tatsächliche Maschine angepasst werden. Auf 64-Bit-Architekturen können die Java-Programme also tatsächlich den vollen Adressraum ausnützen, ohne - wie in herkömmlichen Interpretern - auf 32-Bit-Adressen beschränkt zu bleiben.

Zusätzlich zum Compiler selbst enthält CACAO das notwendige Laufzeitsystem für eine Java Virtual Machine:
Speicherverwaltung (Garbage Collection) und I/O.

Die derzeitige Implementierung läuft auf einem Alpha - Prozessor und unterstützt alle Instruktionen der JavaVM.

Abstract

This thesis documents the implementation of CACAO, a compiler which compiles Java-classfiles to native machine code.

First CACAO loads classfiles containing java programs encoded as instructions for a hypothetical processor - the "Java Virtual Machine" into memory. After that it compiles the parts of the program it needs for execution (Just In Time) into native machine code.

This compilation has the effect to speed up the execution by a factor of 2 to 17 in comparison to an interpreter. The longer the execution time of the programs is - in relation to their length - the higher is the speedup achieved by compiling them.

As a second advantage during compilation all datatypes of the JavaVM are adjusted to fit the actual machine, especially the object pointers. That means java-programms can use the full address space on 64-bit-architectures instead of being limited to 32-bit-addresses as they were on conventional interpreters.

In addition to the compiler CACAO also contains the necessary run time system for a Java Virtual Machine:
Memory management (garbage collection) and I/O.

The current implementation runs on an Alpha - processor and supports the complete instruction set of the JavaVM.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Über Java

Die Programmiersprache Java wurde von der Firma SUN Microsystems entwickelt, und wenn man den Gerüchten trauen will, mit der Absicht damit Waschmaschinen zu steuern. Nach ein paar Jahren in denen sich die Sprache weiterentwickelt hat, wurde dann ihr tatsächlicher Verwendungszweck gefunden, und sie wurde von der Welt als DIE Internetsprache gefeiert.

Programmiersprachen gibt es eigentlich ohnehin genug, ob objektorientiert oder nicht, mit und ohne automatische Speicherbereinigung, in jeder Form und Farbe. Der Grund, warum Java überhaupt verwendet wird, ist, dass SUN ein Format für Binärdateien definiert hat, in dem die übersetzten Javaprogramme an alle Welt verteilt werden. Einmal übersetzte Javaprogramme laufen prinzipiell auf jedem Rechner mit Java-Ünterstützung, es gibt in der Sprachspezifikation nämlich keine Systemabhängigkeiten. Ein zweiter Grund ist der, dass das Javasystem den Rechner vor den Javaprogrammen schützt, und gegen Angriffe von Hackern abschirmt (zumindest in der Theorie).

Der praktische Nutzeffekt dieser beiden Eigenschaften ist, dass man in Java geschriebene (und jetzt als binäre Klassenfiles vorliegende) Programme von irgendwo übers Internet laden und auf jedem Rechner laufen lassen kann, ohne sich Sorgen über die Sicherheit machen zu müssen.

Die Syntax der Sprache ist weitgehend von 'C++' inspiriert, sie enthält aber viele zusätzliche Elemente anderer objektorientierter Sprachen wie zum Beispiel [GR83] oder [US87]. Vor allem in Punkto Speichermanagement unterscheidet sie sich stark von C++. In Java muss der Programmierer die angeforderten Speicherbereiche nicht wieder explizit freigeben, sondern das Laufzeitsystem findet die nicht mehr gebrauchten Bereiche von selbst und kann sie so wiederverwenden (Garbage Collection).

Es gibt eine ganze Menge Bücher, in denen die Sprache und die von SUN zur Verfügung gestellte Entwicklungsumgebung genau beschrieben wird, zum Beispiel: [AG96] [Fla96]

1.2 Über CACAO

Der Namenskonvention entsprechend, alle in Zusammenhang mit Java stehenden Projekte entweder nach Getränken oder nach südostasiatischen Inseln zu benennen, haben wir (mein Diplomarbeitsbetreuer Andreas Krall und ich) uns eben für die Bezeichnung CACAO als Namen für unser System entschieden.

CACAO ist ein System, mit dem die systemunabhängige Bytecode - Darstellung der Java-Programme zur Laufzeit in Maschinensprache umgewandelt wird, und zwar genau dann, wenn der ensprechende Programmteil zum ersten Mal verwendet wird (diese Vorgangsweise nennt sich "Just-In-Time Code-Generation"). Die Maschine für die ich die erste Version von CACAO entwickelt habe, ist eine DEC-Alpha (eine von Digital entwickelte 64-Bit-RISC-Architektur), deshalb werden sich alle Beispiele für erzeugten Maschinencode in dieser Arbeit auf diese Implementierung beschränken. Alle RISC-Systeme sind sich aber ohnehin so ähnlich, dass Portierungen auf andere RISC-Rechner nicht allzu schwierig sein sollten.

Die Frage nach dem Sinn eines solchen Übersetzers von der (schon einmal übersetzten) Bytecodedarstellung auf Maschinensprache kann man mit einem einzigen Wort beantworten:

Geschwindigkeit

In bisherigen Java-Interpretern wird der Bytecode Befehl für Befehl interpretiert, der dabei anfallende Oberhead für den Interpreter ist ziemlich groß, eine Beschleunigung um den Faktor 10 ist durchaus möglich.

Normale Wald-und-Wiesen-Anwendungen (wie es zum Beispiel die jetzt überall modernen javaunterstützten WWW-Seiten sind) brauchen natürlich keinen Compiler, weil die Leistung der meisten Rechner für einfache interaktive Programme ausreicht.

Mit dem JavaVM-Compiler ziele ich hautpsächlich auf aufwendigere Programme (Compiler, Ray-Tracer, etc.), die, weil sie in Java programmiert werden können, höchst portabel werden. Die Entwickler müssen dann ihren Quellcode nicht mehr an alle Plattformen anpassen, sondern können ein und das selbe Programm im binären Format an alle Kunden abgeben (unabhängig von der verwendeten Zielmaschine).

Eine zweite Frage, die unwillkürlich auftaucht, ist die: Warum unbedingt erst zur Laufzeit übersetzen und nicht schon vorher? Die Übersetzung kostet (auch wenn der Compiler sehr effizient ist) eine Menge Zeit, die zur Laufzeit des Programmes dazugerechnet werden muss. Wenn man die Übersetzung einmal vorher machen würde, und dann immer wieder das binäre Maschinencodeprogramm lädt, spart man viel Zeit.

In der Tat gibt es Systeme, die genau so arbeiten: Sie übersetzen die Java-Klassenfiles auf C-Programme, die mit einem C-Compiler weiter übersetzt werden können.

Der (meiner Meinung nach) gravierende Nachteil dabei ist das Einführen dieses zusätzlichen Schrittes, und der dabei benötigten Binärdateien, die natürlich wieder verwaltet werden müssen (Übersetzung der Programme, Verwalten des Versionssystems, etc.). Der zweite Nachteil ist, dass dynamisch geladene Klassen (wie etwa die "Applets" im World Wide Web) nicht so ohne weiteres ausgeführt werden können. Der Aufruf des Compilers (man denke: ein C-Compiler ist dann auch noch nötig) ist in der Tat ziemlich aufwendig wenn nicht gar unmöglich.

CACAO hingegen soll sich dem Benutzer gegenüber genau wie ein herkömmlicher JavaVM-Interpreter verhalten, nur mit dem Vorteil, die Programme ungleich schneller abzuarbeiten.

1.3 Das Klassenfile-System

Wie ich schon oben kurz erwähnt habe, werden Javaprogramme von einem Compiler in entsprechende Klassenfiles übersetzt, dabei gilt die Regel, dass für jede Java-Klasse eine eigene Datei erzeugt wird. Daraus folgt natürlich unmittelbar, dass umfangreichere Programme eine Unmenge an Dateien erfordern, dazu kommt dann noch die gesammte offizielle Laufzeitbibliothek von SUN (Was nach meiner Schätzung einem Umfang von 500 Dateien und 1.5 Megabyte entspricht).

Um dieser Menge an Klassenfiles Herr zu werden, sind sie über mehrere Unterverzeichnisse verteilt (zum Beispiel ist die Klasse `java.lang.String` in Verzeichnis `$JAVAHOME/java/lang` als Datei `String.class` zu finden).

In jedem dieser Klassenfiles steht die gesammte für eine Klasse erforderliche Information, also zum Beispiel:

- Verweis auf die Eltern-Klasse
- Tabelle aller Felder
- Tabelle aller Methoden
- Code für die Methoden als Instruktionen für die JavaVM (Java Virtual Machine)

Eine genauere Beschreibung dieses Klassenfile-Formats kann man in der offiziellen Literatur [LY96] finden.

Kapitel 2

Die Java Virtual Machine

2.1 Die JavaVM ist eine Stackmaschine

Der Programmcode, der letztendlich in den Java-Klassenfiles steht, besteht aus Anweisungen für einen virtuellen Prozessor (der "Java Virtual Machine"). Dieser fiktive Prozessor (der in Zukunft von SUN sogar wirklich in Hardware realisiert werden soll), ist eine Stackmaschine, das heißt, bei allen arithmetischen oder sonstigen Operationen werden zuerst alle Operanden auf einen Stack geladen (entweder durch vorherige Operationen erzeugt, oder aus lokalen Variablen gelesen) und dann durch eine Operation "verbraucht" und durch das Ergebnis dieser Operation ersetzt.

Der Vorteil dieser Codierung ist, dass die Programme ziemlich kompakt dargestellt werden können. Ein Byte reicht für einen Befehl oft aus (daher die Bezeichnung Bytecode), nur bei komplizierteren Befehlen werden noch zusätzliche Operanden gebraucht.

Eine genaue Spezifikation aller gültigen Befehle befindet sich in [LY96].

2.2 Die Datentypen der JavaVM

Weil einfache Integer-Arithmetik für die meisten Fälle nicht ausreicht, unterstützt die JavaVM folgende fünf grundlegende Datentypen:

`INT` 32 Bit mit Vorzeichen

`LONG` 64 Bit mit Vorzeichen

`FLOAT` 32 Bit IEEE-Fließkomma

`DOUBLE` 64 Bit IEEE-Fließkomma

`ADDRESS` (systemabhängiges Format)

Alle Datentypen der Quellsprache Java (z.B: `BOOLEAN, CHAR, SHORT`, etc.) werden auf diese Grundtypen zurückgeführt und mit entsprechenden Umwandungsoperationen umgerechnet.

Wie man sieht, sind diese Datentypen verschieden lang (32 Bit für `INT` und `FLOAT`, 64 Bit für `LONG` und `DOUBLE`), SUN hat daher in der Spezifikation der JavaVM das Konzept der "Stack-Slots" erfunden. Jedes Datum eines 32-Bit-Typs verbraucht immer genau einen solchen Platz am Stack, die 64-Bit-Zahlen verbrauchen jeweils 2 Plätze. Für `ADDRESS` reicht ein einzelner Platz am Stack aus (auch wenn die CPU des tatsächlichen Rechners 64-Bit-Adressen verwendet).

Für jeden der obigen Datentypen gibt es unterschiedliche Instruktionen, die jeweils nur mit Daten eines Typs richtig funktionieren. Das heißt, dass zum Beispiel eine `INT`-Zahl nicht einfach als `FLOAT` oder `ADDRESS` verwendet werden kann, sondern vorher mit speziellen Befehlen in das richtige Format umgewandelt werden muss.

Um zu verhindern, dass "gefälschte" Programme auf unerlaubte Systembereiche zugreifen, indem sie zum Beispiel so eine `INT`-Zahl als `ADDRESS` interpretieren, kann ein Java-System das Programm vorher statisch überprüfen. Diese sogenannte "bytecode verification" ist in [LY96] genau beschrieben und muss auf jeden Fall bei Programmen gemacht werden, die von einer nicht vertrauenswürdigen Quelle stammen (Hacker-Angriffe!).

2.3 Die lokalen Java-Variablen

Die Programmiersprache Java verwendet ebenso wie 'C' lokale Variablen für ihre Unterprogramme (die in Java aber Methoden genannt werden). Dabei muss das System schon im vorhinein wissen, wie viel Platz für Variablen jede Methode benötigt. Bei Aufruf der Methode wird dann eine entsprechende Anzahl von Slots angelegt. Auch für die Variablen gilt die Unterteilung in solche, die nur einen Slot belegen, und solche, die zwei brauchen. Die Adressierung dieser Slots erfolgt einfach linear über ihren Index. Zusammengehörige Slots, die gemeinsam einen 64-Bit-Wert enthalten (etwa bei `LONG`-Zahlen) werden mit dem niedrigeren der beiden Indizes angesprochen.

2.4 Exceptions

Bei unerlaubten Aktionen des Programmes (wie etwa Zugriffe über NULL-Zeiger oder Arrayzugriffe außerhalb der Arraygrenzen) soll die JavaVM nicht abstürzen, sondern eine genau definierte Ausnamebehandlung einleiten. Dazu kann der Programmierer sogenannte *Exception Handler* bereitstellen, die sich darum kümmern. Darüberhinaus kann das Programm auch von sich aus Exceptions erzeugen ("werfen"), die vom nächsten Exception-Handler "gefangen" wird.

Beispiel: Java-Source mit Exceptions

```
static void int divide (int a, int b) {
     int c;
     try {
          c = a / b;
          do_something (c);
          return c;
     }
     catch (ArithmeticException e) {
          system.out.println (ärithmetic exception occured");
          return 0;
     }
}
```

Dieses Beispiel stellt eine kleine Java-Methode dar, die zwei Zahlen dividiert und dann noch eine zweite Methode aufruft. Die in Java eingebaute Division kann aber schief gehen, wenn der Divisor 0 ist. Für solche Fälle kann der Programmierer diese Operation in einen `try`-Block einschließen und eine eventuell auftretende Exception mit der `catch`-Klausel abfangen. Außerdem werden durch das `catch` auch alle `ArithmeticExceptions` gefangen, die während des Aufrufes der `do_something`-Methode passieren könnten und nicht schon dort behandelt wurden. Eine ausführliche Beschreibung der Exceptions mit allen Ausnahmen und Regeln findet sich in [AG96].

Der praktische Nutzeffekt der ganzen Sache ist erstens, dass das Programm niemals irgendwelche undefinierten oder unerlaubten Zugriffe auf den Speicher oder sonstige Resourcen machen kann (das schützt erstens gegen Hacker und zweitens gegen viele Programmierfehler) und zweitens, dass dieses Feature oft einen viel eleganteren Programmierstil erlaubt, als Sprachen bei denen Ausnahmesituationen immer explizit durch bedingte Anweisungen abgefragt werden müssen.

Der große Nachteil dabei ist aber, dass viele Laufzeitüberprüfungen notwendig werden, die unter Umständen recht aufwendig sein können. In einer Interpreter-Umgebung fällt das vielleicht nicht so sehr ins Gewicht, aber bei compiliertem Code macht diese Ausnahmebehandlung einen recht ansehnlichen Teil des Aufwandes aus (aus diesem Grund lassen sich bei CACAO viele dieser Checks abschalten, was das Programm zwar unsicher aber dafür schneller macht).

In den nachfolgenden Beispielen lasse ich den Maschinencode, der für solche Ausnahmebehandlungen notwendig wäre, der besseren Verständlichkeit halber, noch weg und erkläre sie später in einem eigenen Kapitel 6.

2.5 Einige JavaVM-Instruktionen

In den weiter unten folgenden Beispielen verwende ich einige sehr gebräuchliche JavaVM-Befehle, die ich hier kurz erläutern will.

Eine Namenskonvention für die Befehle besagt, dass der erste Buchstabe den Datentyp angibt, den der Befehl bearbeitet. Zum Beispiel führt `FLOAD` die selbe Operation wie `ILOAD` aus, nämlich einen Wert aus einer lokalen Variablen auf den Stack schreiben, allerdings mit einem anderen Datentyp (`FLOAT` anstatt `INT`).

`ILOAD` *slot* Lädt eine `INT`-Zahl aus dem Variablen-Slot *slot* und legt sie oben auf den Stack.

`LLOAD` *slot1* Lädt eine `LONG`-Zahl aus den Slots *slot1* und *slot1*+1 und legt sie auf den Stack.

`ALOAD` *slot* Lädt einen Zeiger (`ADDRESS`) aus dem Slot *slot*.

`ISTORE` *slot* Schreibt die `INT`-Zahl, die oben auf dem Stack liegt, zurück in den Variablen-Slot *slot*. Die Zahl wird dabei vom Stack entfernt.

`IADD` Addiert die beiden `INT`-Zahlen, die oben auf dem Stack liegen und ersetzt sie durch ihre Summe.

`ISUB` Wie `IADD`, aber das Ergebnis ist die Differenz der beiden Werte (dabei wird der alleroberste Wert am Stack vom darunterliegenden abgezogen).

`IMUL` Wie `IADD`, aber Multiplikation.

`BIPUSH` *num* Schreibt die konstante `INT`-Zahl *num* auf den Stack.

`IALOAD` Lädt eine `INT`-Zahl aus einem `INT`-Array und schreibt sie auf den Stack. Dabei müssen der Zeiger auf das Array und der Arrayindex vorher auf dem Stack bereitgestellt werden (sie werden bei dieser Operation automatisch von Stack entfernt).

Einige zusätzliche Befehle, die in den Beispielen vorkommen, werden dort an Ort und Stelle erklärt.

2.6 Beispiele für JavaVM-Code

Ich bringe jetzt einige Beispiele von Befehlssequenzen für die JavaVM, zu denen ich auch den ursprünglichen Sourcecode angebe. Der Java-Sourcecode enthält Variablennamen, die aber im Bytecode vollständig durch Slot-Nummern ersetzt werden. Die Zuordnung von Variablen zu Nummern erfolgt dabei durch den

Java-Compiler (`javac`) zur schon zur Übersetzungszeit. Damit hier keine zu große Verwirrung entsteht, schreibe ich bei den `ILOAD` und `ISTORE`-Befehlen dazu, welche Variable dabei tatsächlich gemeint ist.

Alle diese Beispiele werden später noch einmal verwendet, um zu zeigen, welchen Maschinencode CACAO daraus generiert, und wie er das genau macht.

Beispiel 1

Einfache arithmetische Ausdrücke lassen sich ganz problemlos auf die Stackmaschine übersetzten:

Source:

```
c = a + b;
```

Bytecode:

```
ILOAD     0          ; Wert von a laden
ILOAD     1          ; Wert von b laden
IADD                 ; Zahlen addieren und durch Summe ersetzen
ISTORE    2          ; Ergebnis in Variable c schreiben
```

Beispiel 2

Source:

```
c = a + (b * d - 8);
```

Bytecode:

```
ILOAD     0          ; Wert von a laden
ILOAD     1          ; Wert von b laden
ILOAD     3          ; Wert von d laden
IMUL                 ; b*d berechnen
BIPUSH    8          ; Die Zahl 8 auf den Stack
ISUB                 ; b*d - 8 berechnen
IADD                 ; a + (b*d - 8) berechnen
ISTORE    2          ; Ergebnis in Variable c schreiben
```

Beispiel 3

Ebenso wie arithmetische Berechnung werden auch Array-Zugriffe gehandhabt: Zuerst werden alle nötigen Operanden (Zeiger auf Array, Index in das Array) am Stack bereitgestellt, und dann wird die entsprechende Aktion ausgeführt. (Achtung: In Java werden alle Arrays dynamisch am Heap angelegt und müssen explizit über einen Zeiger angesprochen werden. Es gibt zum Handhaben solcher Adressen eine eigene Menge von JavaVM-Operationen, zum Beispiel `ALOAD`).

Source:

```
c = a[i];
```

Bytecode:

```
ALOAD     0          ; Array-Zeiger aus Variable a laden
ILOAD     8          ; Index aus Variable i laden
IALOAD               ; Wert aus dem Array lesen
ISTORE    2          ; den Wert in die Variable c schreiben
```

Beispiel 4

Methodenaufrufe funktionieren ebenfalls mit Hilfe des Stacks. Alle Argumentwerte müssen vor dem Aufruf auf den Stack gelegt werden. Sobald die Methode aufgerufen wird, werden diese Wert vom Stack entfernt und die neu angelegten Variablen-Slots der Methode hineingeschrieben. Wenn die Methode mit der Bearbeitung fertig ist, kann sie einen Rückgabewert liefern, der wieder auf Stack abgelegt wird, und von der aufrufenden Methode verwendet werden kann.
Die hier gezeigte `INVOKESTATIC` - Instruktion braucht eine genaue Beschreibung der aufzurufenden Methode, dazu gehören sowohl der Name, als auch die genaue "Signatur". Die Signatur ist eine Beschreibung der Typen aller Parameter und des Rückgabewertes. In diesem Beispiel verlangt die Methode `test.funktion` 3 `INT`-Zahlen als Parameter und liefert eine `INT`-Zahl zurück).

Source:

```
c = test.funktion(a+7, x,y);
/* Ruft die Methode 'funktion' der Klasse 'test' auf */
```

Bytecode:

```
ILOAD         0          ; Wert von a laden
BIPUSH        7          ; Konstanten Wert 7 laden
IADD                     ; a + 7 berechnen
ILOAD         4          ; Wert von x laden
ILOAD         5          ; Wert von y laden
INVOKESTATIC  test.funktion (INT,INT,INT)INT
                         ; Aufruf durchführen
ISTORE        2          ; Ergebnis in Variable c schreiben
```

Beispiel 5

Java (und daher auch die JavaVM) kennt Vererbung und virtuelle Methoden, also Methodenaufrufe, bei denen das Ziel des Aufrufes erst zur Laufzeit aufgrund des Objekttypes bestimmt wird. (was mit Hilfe von "Virtual Function Tables" recht einfach zu implementieren ist. Näheres siehe 4.)

Die Instruktion `INVOKEVIRTUAL` führt so einen Aufruf durch. Dabei muss das für die Bestimmung des Zieles notwendige Objekt ebenso am Stack bereitgestellt werden, wie die anderen Argumente. Tatsächlich kann man den Objektzeiger (den 'this'-Zeiger) einfach als 0-ten Parameter für den Aufruf behandeln.

Source:

```
mtype m;                    /* Variable m ist vom Typ 'mtype' */
...                         /* irgendwie bekommt m vorher einen Wert */
c = m.compute(a)+100;       /* Methode 'compute' aufrufen */
```

Bytecode:

```
ALOAD           6           ; Wert von m laden
ILOAD           0           ; Wert von a laden
INVOKEVIRTUAL   mtype.compute (INT)INT
                            ; virtuelle Methode aufrufen
BIPUSH          100         ; konstanten Wert 100 laden
IADD                        ; Summe berechnen
ISTORE          2           ; Ergebnis in Variable c schreiben
```

Beispiel 6

Das Gegenstück zu einem Methodenaufruf ist natürlich die aufgerufene Methode selbst, und die Art und Weise, wie sie an die übergebenen Argumentwerte herankommen und einen Rückgabewert liefern kann. Die übergebenen Werte kommen nämlich als Initialisierungswerte in die neu angelegten Variablen-Slots, beginnend beim Slot 0. (Die Zuordnung im Beispiel wäre also: this = slot 0, a = slot 1). Mit dem Befehl `IRETURN` kann die Methode die Bearbeitung beenden und einen `INT`-Wert als Ergebnis zurückgeben (es gibt einen entsprechenden Befehl für alle Typen, und auch ein `RETURN`, das gar keinen Wert zurückgibt).
In diesem Beispiel zeige ich außerdem eine Anwendung von `GETFIELD`, der Instruktion, mit der die Attributwerte von Objekten gelesen werden können. Für diesem Befehl muss vorher der Zeiger auf das Objekt am Stack bereitstehen, der dann durch `GETFIELD` verbraucht und durch den Wert des Attributes ersetzt wird.

Source:

```
class mtype {              /* Definition der Klasse 'mtype' */
     int value;            /* ein Objekt-Attribut */
     int compute(int a) {  /* virtuelle Methode 'compute' */
          return value*a
     };
```

Bytecode:

```
ALOAD     0                ; Den 'this'-Zeiger laden
GETFIELD  mtype.value      ; Das Attribut value aus diesem Objekt holen
ILOAD     1                ; Wert vom Parameter a laden
IMUL                       ; Multiplikation
IRETURN                    ; Ergebniswert zurückgeben
```

Beispiel 7

Selbstverständlich unterstützt die JavaVM auch bedingte und unbedingte Sprungbefehle um Verzweigungen und Schleifen realisieren zu können. Es gibt im echten JavaVM-Code natürlich keine expliziten Sprunglabels, sondern nur fortlaufende Befehlsadressen, aber ich habe zur besseren Lesbarkeit hier Labels hingeschrieben.

Source:

```
for (i=0; i<10; i++)
          a[i] = b[i]*c;
i=99;
```

Bytecode:

```
     ICONST_0           ; Lädt konstanten Wert 0
     ISTORE      8      ; schreibt diesen Wert in Variable i
     GOTO        l2     ; Sprung zum Befehl l2
l1:  ALOAD       0      ; Lädt den Zeiger aus Variable a
     ILOAD       8      ; Lädt den Wert von i
     ALOAD       1      ; Lädt den Zeiger aus Variable b
     ILOAD       8      ; Lädt den Wert von i
     IALOAD             ; Holt den i-ten Eintrag aus dem Array b
     ILOAD       2      ; Lädt den Wert von c
     IMUL               ; berechnet b[i] * c
     IASTORE            ; Schreibt Ergebnis an i-te Stelle im Array a
     IINC        8,1    ; Erhöht die Variable i um 1
l2:  ILOAD       8      ; Lädt den Wert von i
     BIPUSH      10     ; Lädt den konstanten Wert 0
     IF_ICMPLT   l1     ; springt zu l1, wenn i < 10
     BIPUSH      99     ; Lädt den konstanten Wert 99
     ISTORE      8      ; schreibt diesen Wert in i
```

Kapitel 3

Maschinencode

Wie eingangs erwähnt, soll das System CACAO die JavaVM-Instruktionen nicht während der Laufzeit Schritt für Schritt abarbeiten, sonderen gleich einen Maschinencode für die Zielmaschine (in diesem Fall für einen Alpha-Prozessor) erzeugen, der dann anschließend mit entsprechender Geschwindigkeit ausgeführt werden kann. Dabei darf die Übersetzung selbst aber nicht allzuviel Zeit verbrauchen, weil sie ja selbst zur Laufzeit des Programmes passiert, so dass viele ausgefeilte Optimierungstechniken gar keinen Sinn machen, weil sie nur mit großem Aufwand durchführbar sind (quadratisch oder größer). Ich habe den Compiler daher so einfach wie möglich gehalten, und nur einige wenige (aber doch sehr wichtige) Optimierungen eingebaut.

3.1 Einige Worte über die Zielmaschine

Die Alpha, für die ich die erste Version von CACAO entwickelt habe, ist eine RISC-Maschine mit einem traditionellen Load/Store - Befehlssatz. Sie besitzt 32 Integer- und 32 Fließkommaregister mit einer Breite von jeweils 64 Bit. Mit diesen Registern lassen sich alle für Java notwendigen Datentypen problemlos implementieren. Leider besitzt die Alpha keine eingebaute Integer-Division, für diese Operation muss also ein Unterprogramm aufgerufen werden.

Eine vollständige Diskussion des ganzen Befehlsvorrates ist an dieser Stelle natürlich unmöglich, ich möchte hier auf die entsprechenden Referenzhandbücher verweisen [Cor92] [RS95] [Bha95].

3.2 Aufrufschema von Unterprogrammen

Ich habe den die Methodenaufrufe in CACAO so implementiert, dass die Aufrufskonventionen auf der Alpha ziemlich die selben sind, wie für normale C-Funktionen auch.

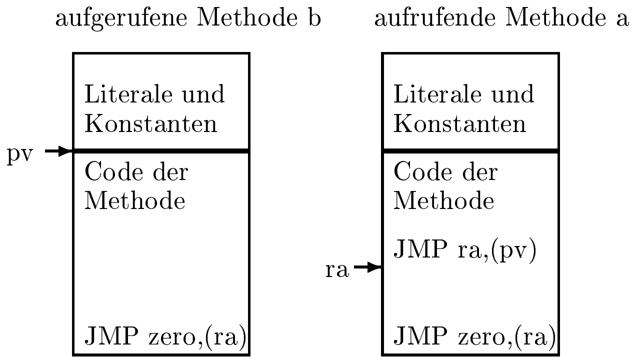
Das Schema funktioniert so:

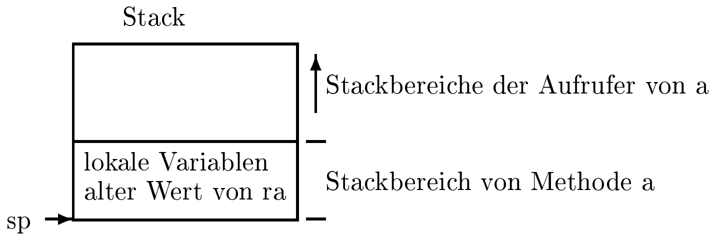
- Bevor ein Unterprogramm aufgerufen wird, muss seine Adresse ins Register `pv` geschrieben werden.
- Die Argumentwerte für die Funktion werden in den Registern `par0` - `par5` bereitgestellt. (Bei mehr als 6 Parametern müssen die restlichen Argumente am Stack übergeben werden).
- Der tatsächliche Aufruf erfolgt mit
  `JMP ra,(pv)`.
  Dieser Befehl schreibt die Rücksprungadresse ins Register `ra` und springt dann zur gewünschten Routine .
- Sobald dieses Unterprogramm seine Arbeit beendet hat, kehrt es mit
  `JMP zero,(ra)`
  zum Aufrufer zurück.

Damit die Unterprogramme ihre Daten temporär ablegen und Registerinhalte sichern können, wird vom Betriebssystem ein Stack bereitgestellt, der mit dem Register `sp` adressiert werden kann. Dieses Register zeigt immer auf die jeweils unterste Grenze des Stacks (dieser Stack wächst von oben nach unten).

Weil es bei jedem Aufruf verpflichtend ist, im Register `pv` die Adresse des Unterprogrammes einzutragen, kann das Unterprogramm dieses Register als Basis verwenden, um irgendwelche notwendigen Konstanten und Literale zu adressieren (Bei 'C'-Programmen, wird zu diesem Zweck normalerweise das Register `gp` benutzt, das vorher explizit geladen werden muss).

Zum besseren Verständis, hier eine kleine Skizze, in der die Situation unmittelbar nach einem Aufruf dargestellt ist:





Die aufrufende Methode a hat hier allerdings das kleine Problem, dass das Register `pv`, mit dem sie selbst bisher den Bereich ihrer eigenen Literale ("Datensegment") adressiert hat, bei der Rückkehr nicht mehr gültig ist. Dieses Problem lässt sich aber sofort lösen, indem die Methode a mit Hilfe der Rückkehradresse in `ra` (deren relative Position zum Anfang der Methode bekannt ist) den Originalzustand des Registers `pv` rekonstruiert (einfach ein `LDA pv,-...(ra)`).

3.3 Maschinencode für die Beispiele

Anhand der Beispiele für JavaVM-Code im vorigen Kapitel möchte ich hier demonstrieren, welchen Maschinencode CACAO daraus generieren kann. Den genauen Algorithmus, nach dem das System vorgeht, beschreibe ich dann etwas später.

Ein paar der wichtigsten Eigenschaften will ich aber doch vorwegnehmen:

- Die Verwaltung des Stacks wird zu Übersetzungszeit in Zugriffe auf temporäre Variablen umgewandelt
- Soweit als möglich werden CPU-Register verwendet, um diese Werte beziehungsweise die Werte der lokalen Java-Variablen aufzunehmen.
- Es werden unter Umständen mehrere JavaVM-Operationen zu einer einzigen Maschineninstruktion zusammengefasst.

Im Maschinencode müssen als Operanden natürlich Register stehen. Die Zuordnung der CPU-Register zu den Operanden der JavaVM und den lokalen Variablen wäre nach zweimaliger Übersetzung (Java-Source -> Java-Bytecode -> Maschinencode) nicht mehr so einfach nachvollziehbar. Ich schreibe statt dessen hier symbolische Registernamen (=Pseudoregister), damit die Beispiele leichter zu lesen sind.

Beispiel 1 (Seite 13)

Weil die CPU 3-Adress-Operationen besitzt, können hier die Stackload- und -store-operationen vollständig eliminiert und alles zu einer einzigen Operation zusammengefasst werden.

Source:
```
c = a + b;
```

Maschinencode:
```
ADDL a,b,c          ; 32-Bit-Addition: c = a+b
```

Beispiel 2 (Seite 13)

Source:
```
c = a + (b * d - 8);
```

Maschinencode:
```
MULL       b,d,tmp0       ; 32-Bit-Multiplikation: tmp0 = b * d
ADDL       0,#8,tmp1      ; Den Wert 8 in tmp1 laden
SUBL       tmp0,tmp1,tmp2 ; tmp2 = tmp0 - tmp1
ADDL       a,tmp2,c       ; Ergebnis berechen: c = a + tmp2
```

Beispiel 3 (Seite 13)

Wie schon erwähnt, sind Arrays in Java ebenfalls Objekte am Heap. Sie haben also ebenso wie Objekte einen kurzen Header mit den wichtigsten Informationen über ihren Typ und über die Größe des Arrays. In diesem Beispiel nehme ich die Größe dieses Headers mit 16 Byte an.

Source:
```
c = a[i];
```

Maschinencode:
```
ADDQS4     i,a,tmp0       ; Adresse im Array berechnen: tmp0 = a + i*4
LDL        c,16(tmp0)     ; INT-Wert aus dem Array laden
```

Beispiel 4 (Seite 14)

Source:

```
c = test.funktion(a+7, x,y);
/* Ruft die Methode 'funktion' der Klasse 'test' auf */
```

Maschinencode:

```
ADDL    zero,#7,tmp0         ; Schreibt den Wert 7 in Register tmp0
ADDL    a,tmp0,par0          ; Addition: par0 = a + tmp0
BIS     x,x,par1             ; Kopieren: par1 = x
BIS     y,y,par2             ; Kopieren: par2 = y
LDQ     pv,funk_zeiger(pv)   ; Lädt den Zeiger auf die Methode
JMP     ra,(pv)              ; Methode anspringen
LDA     pv,-...(ra)          ; Zeiger PV restaurieren
BIS     result,c             ; Rückgabewert in c speichern
```

Hier zeigt sich sehr schön die Notwendigkeit, Parameterwerte beim Aufruf von Methoden in genau definierten Registern zu übergeben (Auf der Alpha sind das zum Beispiel die Register par0 (=R16) bis par5 (=R21) für die ersten 6 Parameter) und den Rückgabewert ebenfalls in einem Register zu empfangen (Register result (=R0)). Die Rolle, die bei den Aufrufkonventionen die Register PV und RA spielen, ist für diese ersten Überlegungen noch nicht von Bedeutung. Zu beachten ist hier aber, dass ich das Register PV auch als Basisregister für das Datensegment benutze, das alle Literale und Konstanten enthält.

Beispiel 5 (Seite 14)

In diesem Beispiel wird, im Gegensatz zum vorigen, die Adresse des Sprungzieles aus einer Tabelle ("Virtual Function Table") geladen und ist abhängig vom tatsächlichen Typ des Objektes `m`. Für jede Klasse existiert in System eine eigene Tabelle, und jedes Objekt einer Klasse beinhaltet den Zeiger auf diese Tabelle.

Source:

```
mtype m;                          /* Variable m ist vom Typ 'mtype' */
...                               /* irgendwie bekommt m vorher einen Wert */
c = m.compute(a)+100;             /* Methode 'compute' aufrufen */
```

Maschinencode:

```
BIS m,m,par0                     ; Kopieren: par0 = m
BIS a,a,par1                     ; Kopieren: par1 = a
LDQ tmp0,0(m)                    ; Zeiger auf Virtual Function Table
LDQ pv,vftbl_offset(tmp0)        ; Eintrag aus der Tabelle laden
JMP ra,(pv)                      ; Methode anspringen
LDA pv,-...(ra)                  ; Zeiger PV restaurieren
ADDL zero,#100,tmp1              ; konstanten Wert 100 in tmp1 schreiben
ADDL result,tmp1,c               ; Addition: c = result + tmp1
```

Beispiel 6 (Seite 15)

Die Methode 'compute' in diesem Beispiel ist eine sogenannte 'leaf'-Methode, das heisst, sie ruft keine anderen Methoden mehr auf. Aus diesem Grund braucht die Rücksprungadresse (im Register RA) nicht woanders gesichert werden.

Source:

```
class mtype {                     /* Definition der Klasse 'mtype' */
     int value;                   /* ein Objekt-Attribut */
     int compute(int a) {         /* virtuelle Methode 'compute' */
          return value*a
          }
     };
```

Maschinencode:

```
LDS  tmp0,offset(par0)            ; Attributwert in temp0 laden
ADDL tmp0,par1,result             ; Rückgabewert berechnen
JMP  zero,(ra)                    ; Methodenbeendigung
```

Beispiel 7 (Seite 16)

Source:

```
for (i=0; i<10; i++)
          a[i] = b[i]*c;
i=99;
```

Maschinencode:

```
     ADDL       zero,#0,i       ; Zahl 0 in Register i schreiben
     BRA        l2              ; Sprung zu Adresse l2
l1:  ADDQS4     i,b,tmp0        ; Adresse im Array b: tmp0 = i*4 + b
     LDL        tmp1,16(tmp0)   ; Wert aus dem Array b lesen
     MULL       tmp1,c,tmp2     ; Multiplikation: tmp2 = tmp1 * c
     ADDQS4     a,i,tmp3        ; Adresse im Array a: tmp2 = i*4 + a
     STL        tmp2,16(tmp3)   ; Wert ins Array a schreiben
     ADDL       zero,#1,tmp4    ; Zahl 1 ins Register tmp4 schreiben
     ADDL       tmp4,i,i        ; Register i um 1 erhöhen
l2:  ADDL       zero,#10,tmp5   ; Wert 10 ins Register tmp5 schreiben
     SUBL       i,tmp5,tmp6     ; Register i mit 10 vergleichen
     BLT        tmp6,l1         ; Sprung, wenn i < 10
     ADDL       zero,#99,i      ; Schreibt 99 in Register i
```

In diesem Beispiel wird offenbar, dass für längere Code-Sequenzen eine ganze Menge temporärer Register notwendig werden. Glücklicherweise haben diese Register normalerweise nur einen sehr kurzen Lebensbereich, so dass man im Endeffekt mit einigen wenigen tatsächichen CPU-Registern auskommen kann indem man sie mit mehreren dieser "Pseudoregister" belegt (Register Allocation).

Kapitel 4

Aufruf virtueller Methoden

4.1 “Normale“ virtuelle Methoden

Es war und ist bei objektorientierten Programmiersprachen eines der Hauptprobleme, virtuelle Methoden (also Unterprogramm-Aufrufe, deren tatsächliches Ziel erst zur Laufzeit ermittelt wird) schnell aufzurufen. Für diesen Zweck existiert die JavaVM - Instruktion `INVOKEVIRTUAL`, deren Verwendung ich in den obigen Beispielen schon kurz erklärt habe.

Bei Systemen mit einfacher Vererbung (jede Klasse kann Methoden nur von einer einzigen Oberklasse erben) wird das Problem normalerweise mit Hilfe einer einfachen ”Virtual Function Table” gelöst [VH96] [KD95]. Dieses Verfahren sollte zwar hinlänglich bekannt sein, ich bringe aber trotzdem ein Beispiel dafür, mit welcher Datenstruktur ich es in CACAO implementiert habe.

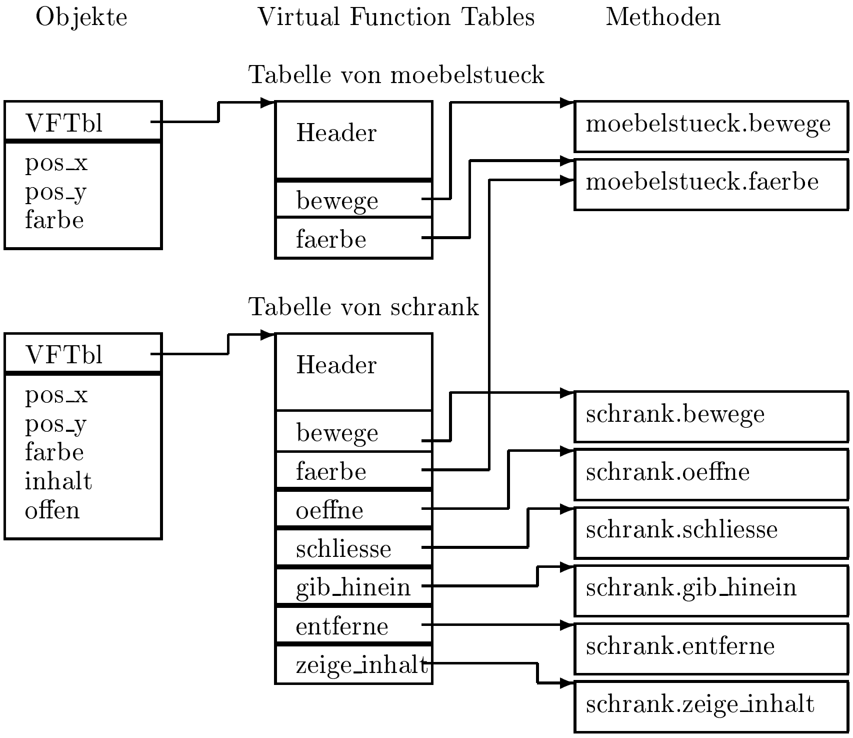
Fragmente eines Beispielprogrammes:

Hier sind zwei Klassen definiert. Die Klasse `kasten` ist von `moebelstueck` abgeleitet und definiert zusätzliche Attribute und Methoden. Dabei überschreibt sie außerdem eine der Methoden von `moebelstueck`. Die `implements`-Klausel ist für dieses Beispiel noch nicht von Bedeutung.

```
class moebelstueck {
     int pos_x,pos_y;
     String farbe;
     void bewege (int x, int y) { ... }
     void faerbe (String f) { ... }
     }
```

```
class kasten extends moebelstueck
implements behaelter, tuer {
     Object[] inhalt;
     boolean offen;
     void bewege (int x, int y) { ... }
     void oeffne () { ... }
     void schliesse () { ... }
     void gib_hinein (Objekt o) { ... }
     void entferne (Objekt o) { ... }
     void zeige_inhalt () { ... }
     }
```

Interne Datenstruktur von CACAO:



In jedem Objekt steht zusätzlich zu den definierten Objekt-Feldern noch ein Zeiger auf die der Klasse des Objekts zugehörige Virtual Function Table. In dieser Tabelle sind alle Methoden verzeichnet, die diese Klasse unterstützt. Wenn eine Klasse von einer Oberklasse abgeleitet wird, dann erbt sie zuerst einmal deren Methoden (die Virtual Function Table wird also einfach kopiert, ohne aber den Code der Methoden selbst zu vervielfältigen). Dann können neue Methoden hinzugefügt oder schon bestehende Methoden "überschrieben" werden (selbstverständlich wird dabei nur die neue Tabelle der Unterklasse modifiziert, die Oberklasse bleibt unverändert).

Im Header der VFTbl stehen noch ein paar weitere Zeiger, wie zum Beispiel ein Zeiger auf zusätzliche Klasseninformation oder die Tabelle für die Interface-Methoden.

Ein Methodenaufruf geht prinzipiell so vor sich: Der Zeiger auf das Objekt ist im Programm verfügbar (in einem Register oder sonstwo schnell erreichbar). Von dort, wo der Zeiger hinzeigt, liest das Programm den Zeiger auf die Virtual Function Table (erster Speicherzugriff). Dann holt das Programm den entsprechenden Eintrag (zweiter Speicherzugriff) aus der Tabelle (der Platz in der Tabelle, an der die Methode mit einer bestimmten Signatur steht, ist schon zur Übersetzungszeit bekannt) und springt an die angegebene Adresse (ein Sprungbefehl).

In Alpha-Code sieht das etwa so aus:

```
LDQ  R28,0(par0)    ; Zeiger auf VFTbl laden
LDQ  pv,offset(R28) ; Zeiger auf die Methode: der Offset
                    ;   ist bei den VFTbl aller
                    ;   abgeleiteten Klassen der selbe.
JMP  ra,(pv)        ; Methode aufrufen
```

4.2 Interface-Methoden

Allerdings beschränkt sich Java nicht nur auf diese Variante mit einfacher Vererbung. Zusätzlich ermöglicht es die Sprache "Interfaces" zu implementieren. So ein Interface ist einfach eine Menge von Methoden. Jede Klasse, die ein Interface implementieren soll, muss dann mindestens alle diese Methoden bereitstellen. Problematisch ist die Sache nur deswegen, weil jede Variable vom Typ des Interfaces (ein Interface ist in Java als eine Art abstrakte Klasse definiert) jedes beliebige Objekt enthalten kann, dessen Klasse das Interface implementiert. Jede Klasse kann darüberhinaus eine beliebige Anzahl von Interfaces bereitstellen. In letzter Konsequenz sind die Interfaces von Java fast ebenso mächtig wie mehrfache Vererbung ("Multiple Inheritance").

In JavaVM-Code werden diese Aufrufe mit dem Befehl `INVOKEINTERFACE` gemacht. Diese Instruktion hat als zusätzliche statische Information die Interface-Klasse und den Namen und die Signatur der aufzurufenden Methode.

Die Datenstruktur, die CACAO verwendet, um die Methoden rasch aufzufinden, stelle ich auch hier wieder mit einem Beispiel dar.

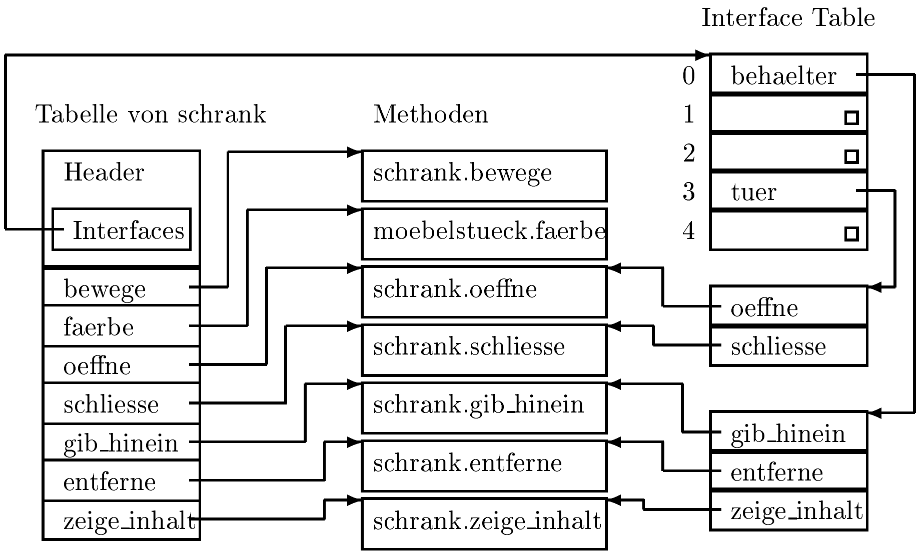
Beispiel für Interfaces:

Hier sind zwei Interfaces definiert. Die Klasse `kasten` im obigen Beispiel implementiert diese Interfaces. Obwohl das auf normale Methodenaufrufe keine Einfluss hat, kann zusätzlich die `INVOKEINTERFACE`-Instruktion mit Objekten vom Typ `kasten` ausgeführt werden.

```
interface tuer {
     void oeffne ();
     void schliesse ();
     }

interface behaelter {
     void gib_hinein (Objekt o);
     void entferne (Objekt o);
     void zeige_inhalt ();
     }
```

Interne Datenstruktur von CACAO:



In der Virtual Function Table steht ein zusätzlicher Eintrag, der auf eine Tabelle zeigt, die für jedes im System vorhandene Interface einen Eintrag enthält. (Interfaces werden in Java anscheinend doch nicht so häufig verwendet, dass diese Tabelle allzugroß werden würde). In allen Interfaces, die die Klasse implementiert, ist ein Zeiger auf eine eigene Methodentabelle eingetragen. Diese Methodentabellen ähneln sehr der einen Tabelle für die "normalen" virtuellen Methoden, mit dem Unterschied, dass jede so viele Einträge hat, wie das zugehörige Interface Methoden bereitstellt.

Ein Aufruf einer Interface-Methode benötigt hier leider zwei Speicherzugriffe mehr als ein Aufruf einer "normalen" Methode. Solche Aufrufe sollten aber relativ selten sein, und dann dürfte das nicht mehr allzusehr ins Gewicht fallen.

Ein Aufruf einer Interface-Methode im Code für die Alpha:

```
LDQ  R28,0(par0)    ; Zeiger auf VFTbl laden
LDQ  R28,...(R28)   ; Zeiger auf die Interface Table
LDQ  R28,...(R28)   ; Zeiger auf die Methodentabelle
LDQ  pv,...(R28)    ; Zeiger auf die Methode selbst
JMP  ra,(pv)        ; Methode aufrufen
```

Als Anmerkung möchte ich sagen, dass es prinzipiell schon möglich wäre, die Interface-Tabelle direkt in die Virtual Function Table zu integrieren (damit könnte man dann ein `LDQ` pro Aufruf einsparen). Dazu bräuchte man aber die Möglichkeit, in einer Speicherstruktur gleichzeitig zwei Arrays unbestimmter Länge einzutragen: Eines nach unten offen, und eines nach oben. In der Sprache 'C' kann man zwar solche nach oben offenen Strukturen definieren, aber immer nur eines gleichzeitig pro Speicherelement. Und weil ich den CACAO fast vollständig in dieser Sprache implementiert habe, müssen auch die erzeugten Datenstrukturen und Tabellen in 'C' darstellbar sein.

Kapitel 5

Übersetzungsalgorithmus

5.1 Phasen

Die Übersetzung des JavaVM-Codes in Maschinencode für die Zielmaschine erfolgt durch CACAO in mehreren Phasen:

- Bestimmen der Superblöcke:
  Hier wird der JavaVM-Code einer Methode einmal durchsucht, um alle Sprungbefehle und deren Sprungziele herauszufinden. Jede JavaVM-Instruktion, die ein solches Sprungziel ist, beginnt einen neuen Superblock. In den anschließenden Phasen werden nur mehr die einzelnen Superblöcke für sich alleine behandelt.

- Zwischencode mit Pseudoregistern erzeugen:
  In diesem Durchgang werden alle Superblöcke einzeln durchgegangen und für jeden eine Darstellung in Zwischencode (eine Art Quadrupelcode mit Erweiterungen) erzeugt. Diese Instruktionen erhalten als Operanden Pseudoregister. Diese Darstellung ist noch unabhängig von der Zielmaschine, hat aber gegenüber dem JavaVM-Code den Vorteil, dass der Stack und die Zugriffe auf die lokalen Variablen bereits völlig aufgelöst, und alle Instruktionsoperanden explizit angegeben sind. Die Methodenaufrufe sind aber nach wie vor durch eine einzige Anweisung dargestellt und noch nicht durch Registeroperationen und Sprungbefehle ersetzt.

- Optimierung:
  In dieser Phase wird der Zwischencode noch ein wenig optimiert, vor allem werden hier sehr viele unnötige Kopieroperationen eliminiert. (In der gegenwärtigen Implementierung von CACAO ist diese Phase mit der vorigen Phase zeitlich verzahnt, so dass eigentlich sofort optimierter Zwischencode erzeugt wird, dennoch sind sie aber als zwei logisch vollständig getrennte Durchgänge zu verstehen)

- Registerbelegung:
Hier werden den oben erzeugten Pseudoregistern tatsächliche CPU-Register zugeordnet. Der Registerbelegungsalgorithmus, den ich bei CACAO verwende ist sehr einfach und schnell (wenn er auch nicht unbedingt eine optimalen Ausnutzung der Register bewirkt).

- Codeerzeugung:
Nach erfolger Registerbelegung wird dann der tatsächliche Maschinencode erzeugt. Weitere Optimierungen wie "Instruction Scheduling" sind in CACAO (noch) nicht implementiert.

- Nachträgliches Eintragen von Sprungadressen:
Viele Sprungziele stehen zum Zeitpunkt des Generierens der Sprungbefehle noch nicht fest und müssen deshalb nachträglich in den vorher erzeugten Code eingetragen werden (Backpatching). Bei der Alpha sind (wie wahrscheinlich auf jeder RISC-Maschine) alle relativen Sprungbefehle ganz gleich aufgebaut, Backpatching ist hier also kein Problem.

5.2 Superblöcke

Superblöcke, wie ich sie im Zusammenhang mit CACAO verstehe, sind alle Sequenzen von JavaVM-Instruktionen, in die von nirgendwo hineingesprungen wird. Ein Verzweigungsbefehl oder ein Methodenaufruf selbst unterbrechen dabei einen Superblock *nicht*, aber an jeder Stelle, wo hingesprungen wird, dort fängt ein neuer Superblock an.

Diese Aufteilung unterscheidet sich etwas von der üblichen Einteilung in Grundblöcke. Grundblöcke dürfen nur Befehle enthalten, die immer geradlinig von oben bis unten abgearbeitet werden, ohne eine Möglichkeit, aus dieser Sequenz herauszuspringen [ASU86]. Das erlaubt zwar viele Optimierungstechniken auf Ebene der Grundblöcke, macht es aber schwieriger Optimierungen zum Beispiel über einen Methodenaufruf hinweg durchzuführen.

In der JavaVM kommt es aber öfters vor, dass Daten auch bei Methodenaufrufen noch am JavaVM-Stack liegen bleiben müssen. Ein Superblock, in dem der Aufruf irgendwo in der Mitte enthalten ist und nicht nicht nur am Ende, erlaubt es hier, Kopierbefehle zwischen Registern noch etwas länger zu verzögern (mit der Hoffnung, sie später vielleicht gänzlich wegzuoptimieren).

Ich will die Sache mit den Superblocks anhand des **Beispiel 7** (Seite 16) besser demonstrieren:

```
Block 1:
     ICONST_0          ; Lädt konstanten Wert 0
     ISTORE     8      ; schreibt diesen Wert in Variable i
     GOTO       12     ; Sprung zum Befehl 12
```

```
Block 2:
l1:  ALOAD      0    ; Lädt den Zeiger aus Variable a
     ILOAD      8    ; Lädt den Wert von i
     ALOAD      1    ; Lädt den Zeiger aus Variable b
     ILOAD      8    ; Lädt den Wert von i
     IALOAD          ; Holt den i-ten Eintrag aus dem Array b
     ILOAD      2    ; Lädt den Wert von c
     IMUL            ; berechnet b[i] * c
     IASTORE         ; Schreibt Ergebnis an i-te Stelle im Array a
     IINC       8,1  ; Erhöht die Variable i um 1
Block 3:
l2:  ILOAD      8    ; Lädt den Wert von i
     BIPUSH     10   ; Lädt den konstanten Wert 0
     IF_ICMPLT  l1   ; springt zu l1, wenn i < 10
     BIPUSH     99   ; Lädt den konstanten Wert 99
     ISTORE     8    ; Schreibt diesen Wert in die Variable i
```

Der Block 3 enthält hier einen Verzweigungsbefehl, dieser unterbricht den Block selbst aber nicht.

5.3 Pseudoregister

Die Pseudoregister von CACAO haben die Aufgabe, alle temporären Daten, die eine Methode haben kann, aufzunehmen. Sowohl die lokalen Variablen-Slots, als auch die Einträge des JavaVM-Stacks werden auf solche Pseudoregister abgebildet. Ob dann tatsächlich CPU-Register dafür verwendet werden können oder ob die Wert doch wieder in den Speicher ausgelagert werden müssen, stellt sich aber erst in der Register-Allokations-Phase heraus.

Alle Pseudoregister haben einen bestimmten Typ (nämlich einen der schon oben erwähnten Grundtypen: `INT, LONG, FLOAT, DOUBLE, ADDRESS`). Diesen Typ stelle ich in allen Beispielen durch den Namen des Registers dar (zum Beispiel bedeutet `A4`, dass es sich um ein Pseudoregister vom Typ `ADDRESS` handelt, der 4er ist einfach eine fortlaufende Nummerierung).

Der Lebensbereich der Pseudoregister, die in den Zwischencode-Instruktionen als Operanden verwendet werden, ist entweder sehr kurz (bei Operanden von Stack-Operationen), oder aber er geht über die ganze Methode (bei den lokalen Java-Variablen).

Für die erstere Art von Pseudoregistern gilt folgende Definition des Lebensbereichs:

Innerhalb eines Superblockes ist ein Pseudoregister aktiv von der ersten Instruktion, die es als Zieloperand verwendet, bis zum ersten `DROP` (siehe 5.6) mit diesem Register (wenn kein `DROP` erfolgt, dann erstreckt sich die Gültigkeit eben bis zum Ende des Blockes). Zusätzlich kann bei jedem Block eine Menge von Pseudoregistern gleich zu Begin des Blockes aktiv sein (ist notwendig bei blockübergreifenden Aktionen, die den Java-Stack benutzen).

Im Zusammenhang mit Methodenaufrufen muss ich noch erwähnen, dass es diese Pseudoregister in zwei Ausprägungen gibt, nämlich als *gesicherte* und *ungesicherte* Register. Die erste Art (der auf jeden Fall alle Register mit globalem Lebensbereich angehören) überlebt Methodenaufrufe unbeschadet, während die Werte der anderen Register bei Methodenaufrufen zerstört werden dürfen (wird wenn möglich für die kurzlebigen Zwischenergebnisse verwendet). Der Compiler nimmt anfangs alle temporären Pseudoregister als *ungesichert* an, bei jedem Methodenaufruf markiert er aber alle Register als *gesichert*, die während des Aufrufes auf dem Stack (siehe 5.4) liegen bleiben.

5.4 Verwaltung des JavaVM-Stacks

Damit jetzt keine Unklarheit aufkommt, möchte ich gleich hier sagen, dass ich die Bezeichnung "Stack" unter Umständen für zwei verschiedene Sachen benutze. Erstens für den von der JavaVM benötigten Operanden-Stack (JavaVM-Stack) und den Stack, auf dem Unterprogramme ihre lokalen Variablen halten und Registerwerte sichern (also der Unterprogramm-Stack, wie er in zum Beispiel "C" verwendet wird). Normalerweise wird es aus dem Zusammenhang klar sein, was ich meine, in Zweifelsfällen schreibe ich explizit "JavaVM-Stack" wenn ich diesen meine.

Ein Ziel des Compilers ist es, den JavaVM-Stack vollständig durch Zugriffe auf Pseudoregister zu ersetzen. Dazu führt der Compiler, während er alle JavaVM-Instruktionen schrittweise durchgeht, einen Stack von Pseudoregistern mit, die die dem JavaVM-Stack entsprechendenen Werte enthalten. Bei jeder Operation werden entsprechend der Semantik Register vom Stack entfernt, oder neue Pseudoregister angelegt und auf den Stack draufgegeben. Dadurch wird jedes Pseudoregister für den Stack an nur einer einzigen Stelle im Code beschrieben (diese Eigenschaft des Stack entpricht der Static Single Assignment Form wie sie in [CFR+91] beschrieben wird).

Die Struktur des Stacks muss aber nicht zu jeder Instruktion abgespeichert werden (was unter Umständen einen ziemlichen Speicheraufwand bedeuten würde), es reicht aus dass der Compiler sie sich nur für den Anfang eines jeden Blockes merkt.

Weil Werte auf dem Stack von einem Block zum nächsten übergeben werden können, darf der Compiler diese Blöcke nicht in willkürlicher Reihenfolge abarbeiten, sondern muss sich in der Reihenfolge an den Kontrollflussgraphen halten, das heißt, er muss zuerst den ersten Block der Methode (Stelle 0 im Java-Bytecode) übersetzen und dann alle von diesem Block aus angesprungenen Blöcke, und so weiter alle Blöcke, die schon irgendwie erreicht worden sind.

Bei jedem Sprungbefehl muss der Compiler überprüfen, ob der Zielblock schon eine Anfangsstackbelegung hat. Wenn nicht, dann wird die aktuelle Stackbelegung als Anfangsstackbelegung des Zielblockes eingetragen (und der Zielblock als schon erreicht markiert, damit er später übersetzt werden kann).

Wenn aber der Zielblock schon eine gültige Anfangsbelegung hat (oder gar schon vollständig in Zwischencode übersetzt worden ist), dann kann dessen Anfangsbelegung nicht mehr so ohne weiteres geändert werden. Statt dessen wird der gerade aktive Stack so modifiziert, dass er mit der Anfangsbelegung des Zielblockes übereinstimmt. Dazu sind im allgemeinen Fall Kopieroperationen für Pseudoregister notwendig (vgl.: ϕ - *functions* in [CFR+91]).

Glücklicherweise aber ist der JavaVM-Stack bei Superblockgrenzen meistens leer (zumindest bei der derzeitigen Version des Java-Compilers), und wenn nicht, dann stimmt die Struktur der Stacks oft vollständig überein, so dass diese aufwendigen Umladeoperationen tatsächlich recht selten gemacht werden müssen.

5.5 Die lokalen JavaVM-Variablen

Im Gegensatz zum JavaVM-Stack ist die Verwaltung der lokalen JavaVM-Variablen viel einfacher und braucht auch keine Rücksicht auf die zeitliche Abfolge der Instruktionen nehmen.

Ich habe schon gesagt, dass die Variablen-Slots ebenso auf Pseudoregister abgebildet werden, wie die temporären Werten auf dem Stack.

Dazu braucht der Compiler irgendeine Art Übersetzungstabelle, mit der er die Zugriffe auf die Variablen-Slots zu Zugriffen auf Pseudoregister umwandeln kann. Leider muss jedes Pseudoregister (das ja später auf ein einziges CPU-Register abgebildet wird) auf jeden Fall einen fixen Typ haben (`INT`, `LONG`, `FLOAT`, `DOUBLE` oder `ADDRESS`), die Slots können aber jeden beliebigen Wert aufnehmen. Eine eins zu eins - Zuordnung ist damit nicht möglich.

Glücklicherweise ist es in der JavaVM-Spezifikationen verboten, mit einer Instruktion auf Daten eines falschen Typs zuzugreifen (also etwa eine Adresse in Slot 0 speichern, und dann als Integer-Zahl herauslesen). Außerdem ändert sich die Belegung der Variablen normalerweise nicht wesentlich innerhalb einer Methode, sodass ich folgende einfache Vorgangsweise für recht sinnvoll halte:

Für jeden Variablen-Slot und jeden Java-Grundtyp (also in Form eines rechteckigen Matrix) merkt sich der Compiler ein Pseudoregister, das den Wert dieses Variablen-Slots enthält. Ganz zu Anfang der Übersetzung enthält die Matrix noch gar keine Pseudoregister, aber jedesmal, wenn eine Instruktion eine Variable anspricht, wird entweder ein neues passendes Pseudoregister angelegt, oder das bereits vorhandene verwendet.

Ein Beispiel: Die Operation `ILOAD 3`, die eine Integerzahl aus dem Slot 3 auf den Stack lädt, bewirkt folgendes: Der Compiler schaut, ob in der Matrix in der Spalte 3 und in der Zeile `INT` ein Pseudoregister eingetragen ist. Wenn ja, dann verwendet er es für diese Aktion (er generiert einen COPY-Befehl und manipuliert den Stack entsprechend). Wenn noch keines eingetragen ist, dann erzeugt der Compiler eben jetzt ein Pseudoregister vom Typ `INT`, trägt es dort ein und macht dann ebenfalls die entsprechenden Aktionen.

In der Regel bleiben sehr viele Zellen dieser Matrix frei, weil die meisten Variablen-Slots nur einmal belegt werden. In diesem Fall ist in jeder Spalte nur eine Zelle belgegt. Bei Mehrfachbelegung des selben Slots mit Variablen unterschiedlichen Typs können aber auch mehrere Pseudoregister für einen einzigen Stack-Slot notwendig sein (Pseudoregister verschiendenen Typs sind grundsätzlich nicht miteinander kompatibel).

Als weiters Beispiel bringe ich einmal folgende JavaVM-Sequenz, die zwar ziemlich nutzlos ist, aber das Prinzip der Pseudovariablenvergabe gut illustriert. Hier werden ganz einfach irgendwelche Werte in den Variablen-Slots herumkopiert. Diese Werte sollten natürlich vorher schon irgendwie dorthin gekommen sein, aber das ist für dieses Beispiel nicht so wichtig.

```
ILOAD   0       ; Lädt eine INT-Zahl aus Slot 0
ISTORE  1       ; Schreibt die Zahl in Slot 1
FLOAD   3       ; Lädt eine FLOAT-Zahl aus Slot 3
DLOAT   5       ; Lädt eine DOUBLE-Zahl aus Slot 5
DSTORE  3       ; Schreibt die DOUBLE-Zahl nach Slot 3
FLOAT   7       ; Lädt eine FLOAT-Zahl aus Slot 7
FSTORE  3       ; schreibt diese FLOAT-Zahl in Slot 3
FSTORE  0       ; schreibt die FLOAT-Zahl, die vorher in Slot 3 war,
                ; jetzt in den Slot 0
```

CACAO geht diese Sequenz durch und erzeugt währenddessen folgende Übersetzungstabelle:

	INT	FLOAT	LONG	DOUBLE	ADDRESS
0	I0	F0			
1	I1				
2					
3		F3		D3	
4					
5				D5	
6					
7		F7			

Die Pseudoregister sind voneinander unabhängig, das bedeutet auch, dass die `FSTORE 0`-Instruktion den Wert des Pseudoregisters `I0` nicht beeinflussen würde, obwohl sie auf den selben Slot zugreift.

Eine kleine Anmerkung: Damit man die Zuordnung von Pseudoregistern zu den Slots besser im Auge behalten kann, gibt CACAO den Pseudoregistern, die für diesen Zweck erzeugt werden, gleich die Nummer des entsprechenden

Slots (also ist zum Beispiel für den `FLOAT`-Wert auf Slot 3 das Pseudoregister `F3` zuständig).

5.6 Zwischendarstellung als Pseudokommandos

Nach der Unterteilung in Superblöcke geht der Compiler jeden einzelnen Block Befehl für Befehl durch und generiert einen dazu passenden Zwischencode. Dieses Verfahren bildet aber immer nur genau eine JavaVM-Instruktion auf ein oder mehrere Zwischencodeoperationen ab, so dass etwa die im **Beispiel 1** (Seite 13) gegebene `ILOAD / ILOAD / IADD / ISTORE` - Sequenz hier nicht so ohne weiteres auf einen einzigen Zwischencode-Befehl reduziert werden kann. Diese Dinge wird dann die Optimierungsphase erledigen.

Alle Pseudokommandos (so nenne ich die Instruktionen im Zwischencode) haben als Quell- und Zieloperanden Pseudoregister und gegebenenfalls zusätzliche konstante Operanden (Sprungadressen, konstante Zahlenwerte oder Adressen, etc.). Der Stack der JavaVM ist hier bereits vollständig eliminiert und durch Pseudoregister ersetzt, ebenso wie die lokalen Variablen der JavaVM.

Vor weiteren Erklärungen bringe ich gleich einmal einen unoptimierten Zwischencode für **Beispiel 7** (Seite 16) :

```
Block 1, Anfangs-Stack: leer
     LOADCONST (INT)   #0 -> I9                ; ICONST_0
     MOVE (INT)        I9 -> I8                ; ISTORE 8
     DROP              I9
     BRA (GOTO)        Block 3                 ; GOTO 12
```

```
Block 2, Anfangs-Stack: leer
l1:  MOVE (ADDR)       A0 -> A13             ; ALOAD 0
     MOVE (INT)        I8 -> I14             ; ILOAD 8
     MOVE (ADDR)       A1 -> A15             ; ALOAD 1
     MOVE (INT)        I8 -> I16             ; ILOAD 8
     OP2 (IALOAD)      A15,I16 -> I17        ; IALOAD
     DROP              A15
     DROP              I16
     MOVE (INT)        I2 -> I18             ; ILOAD 2
     OP2 (IMUL)        I17,I18 -> I19        ; IMUL
     DROP              I17
     DROP              I18
     OP3 (IASTORE)     A13,I14,I19           ; IASTORE
     DROP              A13
     DROP              I14
     DROP              I19
     MOVE (INT)        I8 -> I20             ; IINC 8 1
     LOADCONST (INT)   #1 -> I21
     OP2 (IADD)        I20,I21 -> I22
     DROP              I20
     DROP              I21
     MOVE (INT)        I22 -> I8
     DROP              I22

Block 3, Anfangs-Stack: leer
l2:  MOVE (INT)        I8 -> I10             ; ILOAD 8
     LOADCONST (INT)   #10 -> I11            ; BIPUSH 10
     BRA (IF_ICMPLT)   I10,I1, Block 2       ; IF_ICMPLT l1
     DROP              I10
     DROP              I11
     LOADCONST (INT)   #99 -> I12
     ; BIPUSH 99
     MOVE (INT)        I12 -> I8             ; ISTORE 8
```

In dieser Zwischendarstellung gibt es nur mehr einige wenige Typen von Instruktionen (`MOVE`, `OP2`, `OP3`, `LOADCONST`, ...), mit denen alle JavaVM-Befehle dargestellt werden. Dazu ist bei vielen Instruktionen noch ein zusätzlicher Parameter notwendig, der die Operation genauer spezifiziert (ich habe sie in obigem Beispiel in Klammern dazugeschrieben).

Derzeit braucht das System CACAO für die Darstellung aller möglichen JavaVM-Befehle folgende Zwischencode-Instruktionen:

`LOADCONST` Schreibt einen konstanten Wert des gewünschten Typs (`INT`, `LONG`, `FLOAT`, `DOUBLE`, `ADDRESS`) in ein Pseudoregister.

`MOVE` Kopiert einen Wert aus einem Pseudoregister in ein anderes.

`OP` n Führt eine (arithmetische oder sonstige) Operation mit n Quellregistern durch und schreibt das Ergebnis in das Zielregister (wenn angegeben). So eine Operation kann eventuell auch auf den Speicher zugreifen (bei Array-Load bzw. -Store-Befehlen), allerdings auf keine anderen Pseudoregister als im Befehl explizit angegeben sind.

`MEM` Dient zum Zugreifen auf den Speicher, allerdings kann hier im Gegensatz zu `OP` ein zusätzlicher Offset spezifiziert werden (für Felder innerhalb von Objekten).

`BRA` Für alle Arten von Sprung- bzw. Verzweigungsbefehlen, inklusive Methodenbeendigung.

`TABLEJUMP` Für Sprünge mit einer Sprungtabelle (wird eher selten gebraucht)

`METHOD` Mit dieser Instruktion werden alle Arten vom Methodenaufrufen getätigt. Sie erlaubt eine unbegrenzte Anzahl von Parametern (müssen vorher in Pseudoregistern bereitgestellt werden) und schreibt den Rückkehrwert der Methode in das angegebene Zielregister.

`DROP` Das ist eigentlich gar keine Instruktion, sondern nur ein Hinweis an den nachfolgenden Registerallokator, dass das angegebene Pseudoregister ab hier nicht mehr gebraucht wird, und das ihm zugewiesene CPU-Register wieder anderwertig verwendet werden kann.

`ACTIVATE` Ist das Gegenstück zu `DROP` und sagt dem Registerallokator, dass ein Pseudoregister ab hier aktiv ist. Normalerweise wird das `ACTIVATE` bei allen Instruktionen, die ein Zielregister haben (`OP, MEM,` etc.) implizit durchgeführt, es gibt aber ein paar Fälle, wo es dennoch notwendig ist (nämlich für die etwas seltsame `JSR/RET` - Konstruktion der JavaVM).

Eine vollständigere Aufstellung der Befehle mit möglichen Parametern findet sich weiter hinten in C.

5.7 Optimierungen

Ich habe schon oben erwähnt dass die mit dem bis hier erklärten Verfahren erzeugten Zwischencode-Sequenzen noch nicht sehr effizient sind. Vor allem deswegen, weil jede JavaVM-Instruktion einzeln für sich übersetzt wird, und auf jeden Fall auch irgendeine Zwischencode-Instruktion erzeugt. Dabei könnten aber die meisten Lade- und Speicheroperationen für die lokalen Variablen vollständig eliminiert werden.

Als Demonstration bringe ich wieder eines meiner bewährten Beispiele, und möchte mit dessen Hilfe mein Optiminierungsverfahren erläutern.

Zwischencode für Beispiel 2 (Seite 13)

```
MOVE (INT)        I0 -> I4            ; ILOAD 0
MOVE (INT)        I1 -> I5            ; ILOAD 1
MOVE (INT)        I3 -> I6            ; ILOAD 3
OP2 (IMUL)        I5, I6 -> I7        ; IMUL
DROP              I5
DROP              I6
LOADCONST (INT)   #8 -> I8            ; BIPUSH 8
OP2 (ISUB)        I7, I8 -> I9        ; ISUB
DROP              I7
DROP              I8
OP2 (IADD)        I4, I9 -> I10       ; IADD
DROP              I4
DROP              I9
MOVE (INT)        I10 -> I2           ; ISTORE 2
DROP              I10
```

5.7.1 Eliminierung der Ladebefehle

Die erste Optimierung zielt darauf, die `MOVEs`, die durch die Ladebefehle von lokalen Variablen entstehen, soweit als möglich zu eliminieren. Der Optimierer arbeitet dazu den Zwischencode schrittweise ab, und entfernt fürs erste einmal jeden `MOVE`-Befehl, auf den er trifft. Er merkt sich aber in einer Datenstruktur alle Pseudoregister, die Kopien von anderen Registern enthalten sollten (es aber noch nicht tun, weil der entsprechnde `MOVE`-Befehl unterdrückt worden ist). Solange auf so ein Register nur lesend zugegriffen wird, kann ganz einfach an dessen Stelle das Register verwendet werden, das den Originalwert enthält. Im günstigsten Fall erreicht der Optimierer irgendwann ein `DROP`, das das kopierte Register ungültig macht, sodass er die `MOVE`-Operation komplett unter den Tisch fallen lassen kann. Im ungünstigen Fall wird das Originalregister aber vorher durch andere Operationen zerstört, und der `MOVE`-Befehl muss schließlich doch noch erzeugt werden.

Optimierter Zwischencode für Beispiel 2 (Seite 13) :

```
                                        ; ILOAD 0
                                        ; ILOAD 1
                                        ; ILOAD 3
OP2 (IMUL)        I1, I3 -> I7          ; IMUL
LOADCONST (INT)   #8 -> I8              ; BIPUSH 8
OP2 (ISUB)        I7, I8 -> I9          ; ISUB
DROP              I7
DROP              I8
OP2 (IADD)        I0, I9 -> I10         ; IADD
DROP              I9
MOVE (INT)        I10 -> I2             ; ISTORE 2
DROP              I10
```

5.7.2 Eliminierung der Speicherbefehle

Wie in diesem Beispiel offenbar wird, kann das obige Verfahren das letzte `MOVE` nicht wegoptimieren, weil am Ende des Blockes das Ergebnis auf jeden Fall im Pseudoregister `I2` vorhanden sein muss (die Optimierungen erfolgen nicht über die Grenzen der Superblöcke hinweg). Allerdings wäre es gar nicht notwendig, das Pseudoregister `I10` überhaupt als Zwischenträger zu verwenden.

Die zweite Optimierung geht den Zwischencode ebenfalls von oben nach unten durch (dass alle Phasen den Code in der selben Richtung durcharbeiten ist deshalb nötig, damit ich sie in meiner Implementierung gleich in einem einzigen Durchgang zeitlich verzahnt durchführen kann). Bei jedem `MOVE`, das unmittelbar von einem passenden `DROP` gefolgt wird (also eine Sequenz: `MOVE x -> y; DROP x`), sucht er die unmittelbar vorher gelegene Instruktion (wobei er allerdings alle `DROP`s ignoriert) und versucht das dortige Operationsziel (wenn es sich um das selbe Pseudoregister `x` handelt), so anzupassen, dass die `MOVE`-`DROP`-Sequenz selbst nicht mehr notwendig ist.

Das obige Beispiel wird also noch etwas verbessert.

Besser optimierter Zwischencode für Beispiel 2 (Seite 13) :

```
                                          ; ILOAD 0
                                          ; ILOAD 1
                                          ; ILOAD 3
OP2 (IMUL)         I1, I3 -> I7           ; IMUL
LOADCONST (INT)    #8 -> I8               ; BIPUSH 8
OP2 (ISUB)         I7, I8 -> I9           ; ISUB
DROP               I7
DROP               I8
OP2 (IADD)         I0, I9 -> I2           ; IADD + ISTORE 2
DROP               I9
```

Dieses Verfahren eliminiert normalerweise die meisten `MOVE`-Operationen und lässt nur mehr die Instruktionen übrig, die tatsächlich etwas berechnen. Algebraische Umformungen oder raffinierte Dead-Code-Elimination haben auf dieser Stufe der Übersetzung (die - ich möche noch einmal daran erinnern - zur Laufzeit passiert) keinen Sinn mehr, sondern müssen schon vorher vom Java-Compiler durchgeführt werden.

5.8 Registerbelegung

Für die Pseudoregister muss der Registerallocator eine Belegung mit tatsächlichen CPU-Registern durchführen. Der Algorithmus darf dabei aber nicht allzu aufwending sein, weil ja die ganze Übersetzung erst zur Laufzeit des Programmes passiert.

Ein globale Registerbelegung nach [Cha82] oder [PBT89] benötigt große Datenstrukturen für den Interferenzgraphen und dementsprechende lange Übersetzungszeiten.

Ich habe deshalb so komplizierte Methoden wie "Graph Coloring" nach gar nicht erst implementiert.

Mein Registerallokator arbeitet ohne irgendeine Kostenberechnung und belegt die Pseuodoregister ganz einfach in der Reihefolge, in der sie im Code auftreten (also nicht unbedingt in der Abfolge, die durch den Kontroll- und Datenflussgraphen gegeben wäre).

Wenn keine CPU-Register mehr übrig sind, dann werden eben Plätze im Speicher vergeben (Register Spilling). Glücklicherweise sind die meisten Java-Methoden relativ kurz, und so liefert diese Vorgangsweise recht gute Ergebnisse (vor allem dann wenn der Java-Compiler schon vorher dahingehend optimiert, dass der JavaVM-Code einen möglichst kleinen Stack aufbaut und möglichst wenige Variablen-Slots benutzt).

Die Vorgangsweise des Allokators ist für jeden Block die folgende:

- Alle Pseudoregister, die am Anfangs des Blocks am Stack liegen (siehe oben, 5.4), werden gleich mit CPU-Registern belegt, oder eine schon vorhandene Belegung wird behalten. Diese Pseudoregister gelten jetzt als aktiv.

- Der Allokator geht den Block schrittweise durch und aktiviert jedes Pseudoregister, das als Zieloperand einer Instruktion dient. Ausgenommen sind natürlich die `DROP`-Instruktionen, hier wird das Pseudoregister deaktiviert.

- Wann immer der Allokator auf ein zu aktivierendes Pseudoregister ohne Belegung trifft, dann sucht er ein zur Zeit freies CPU-Register und verwendet dieses.

- Wenn das Pseudoregister schon vorher einmal eine Belegung erhalten hat, dann versucht der Allokator diese Belegung beizubehalten. Das geht aber nur, wenn das CPU-Register nicht gerade für irgendwelche anderen Pseudoregister verwendet wird.

- Ist das aber der Fall, dann gibt der Allokator dem zu aktivierenden Pseudoregister eine neue Belegung. Der Allokator hat aber keine expliziten Aufzeichnungen darüber, welche Pseudoregister in Programmteilen, die er schon durchgearbeitet hat, gleichzeitig aktiv sind. Damit auf keinen Fall irgendwelche Konflikte auftreten, vergibt er eben ein noch ganz neues CPU-Register, das bisher in der Methode nirgends verwendet worden ist (siehe Beispiel).

Dieser Algorithmus ist aber nur für die temporären Pseudoregister brauchbar, die globalen Register (global im Sinne von: in der ganzen Methode verfügbar) werden ganz am Schluss extra belegt. Dazu wird jedem davon ein ganz "frisches" CPU-Register (damit auch hier auf keinen Fall Konflikte entstehen können) zugeteilt.

Zum besseren Verständnis bringe ich jetzt ein kleines Beispiel. Dieses Programmstück in JavaVM-Code ist zugegebenermaßen etwas konstruiert und würde wahrscheinlich niemals aus irgendeinem Java-Sourceprogramm erzeugt werden, aber es entspricht der Spezifikation und deshalb muss es von CACAO korrekt übersetzt werden können.

Beispiel 8 für Registerbelegung

Bytecode:

```
Block 1:
     ILOAD         4                   ; Variable 4 auf den Stack laden
     IFNE          l1                  ; Sprung wenn 0 -> l1
     ILOAD         0                   ; Wert von Variable 0 auf den Stack laden
     INEG                              ; diesen Wert negieren
     ILOAD         0                   ; Wert von Variable 0
     ILOAD         1                   ; Wert von Variable 1
     ILOAD         2                   ; Wert von Variable 2
     IADD                              ; berechne Var1 + Var2
     IADD                              ; berechne (Var0 + Var1) + Var2
     ISTORE        0                   ; Ergebnis in Variable 0 speichern
     GOTO          l2                  ; (am Stack liegt noch der alte Wert von Var0)
Block 2:
l1:  ILOAD         1                   ; Wert von Variable 1
     ILOAD         0                   ; Wert von Variable 0
     ISTORE        1                   ; diesen Wert auf Variable 1 schreiben
     INEG                              ; alten Wert von Var1 negieren
     INT2BYTE                          ; diesen Wert auf auf 8 Bit trimmen
Block 3:
l2:  ISTORE        5                   ; den Wert vom Stack in Variable 5 schreiben
```

optimierter Zwischencode:

```
Block 1, Anfangs-Stack: leer
     BRA (IFNE)     I4, Block 2        ; ILOAD + IFNE
     OP1 (INEG)     I0 -> I6           ; INEG
     OP2 (IADD)     I0, I1 -> I7       ; ILOAD + ILOAD + IADD
     OP2 (IADD)     I7, I2 -> I0       ; ILOAD + IADD + ISTORE
     DROP           I7
     BRA (GOTO)     Block 3            ; GOTO
Block 2, Anfangs-Stack: leer
     MOVE (INT)     I1 -> I8           ; ILOAD
     MOVE (INT)     I0 -> I1           ; ILOAD + ISTORE
     OP1 (INEG)     I8 -> I9           ; INEG
     DROP           I8
     OP1 (INT2BYTE) I9 -> I10          ; INT2BYTE
     DROP           I9
     MOVE (INT)     I10 -> I6          ; Stackanpassung
Block 3, Anfangs-Stack: I6
     MOVE (INT)     I6 -> I5           ; ISTORE
```

Es gibt in diesem Beispiel nur 5 temporäre Pseudoregister: `I6, I7, I8, I9, I10`. Die anderen Pseudoregister enthalten die Werte lokaler Variablen und werden gesondert belegt. Die Belegung mit tatsächlichen CPU-Registern geht schrittweise vor sich und zwar laut dem obigen Algorithmus.

Ich zeige in dieser Tabelle, welche Belegungen der Compiler schrittweise für die Pseudoregister findet, indem er den Zwischencode Instruktion für Instruktion durcharbeitet und jedem neuen Pseudoregister ein freies CPU-Register zuordnet. Wenn der Compiler damit fertig ist (unterste Zeile), dann hat er eine korrekte Belegung der Register gefunden.

	I6	I7	I8	I9	I10	aktive Reg.
Block 1	-	-	-	-	-	-
... -> I6	R0	-	-	-	-	R0
... -> I7	R0	R1	-	-	-	R0, R1
DROP I7	R0	R1	-	-	-	R0
Block 2	R0	R1	-	-	-	-
... -> I8	R0	R1	R0	-	-	R0
... -> I9	R0	R1	R0	R1	-	R0, R1
DROP I8	R0	R1	R0	R1	-	R1
... -> I10	R0	R1	R0	R1	R0	R0, R1
DROP I9	R0	R1	R0	R1	R0	R0
... -> I6	R2	R1	R0	R1	R0	R0, R2
Block 3	R2	R1	R0	R1	R0	R2
Fertig	R2	R1	R0	R1	R0	-

Das einzige Problem, auf das der Allokator hier stößt ist, dass das Pseudoregister I6 als Ziel einer Operation (`MOVE I10->I6`) dient, obwohl es schon eine Belegung hat. Ungünstigerweise wird das Register `R0` gerade für andere Zwecke benötigt, deshalb muss dem `I6` das bisher unbenutzte Register `R2` zugewiesen werden. (Eine Belegung mit `R1`, das gerade frei wäre, kommt nicht in Frage, weil es dabei zu einem Konflikt zwischen `I6` und `I7` kommen würde).

5.8.1 Register-Spilling

Bei größeren Methoden reichen die CPU-Register oft nicht aus. Dann muss der Allokator gewisse Pseudoregister auf den Speicher auslagern. Diese ausgelagerten Pseudoregister werden im logischen Konzept genauso behandelt, wie die normalen Register. Der Zwischencode sieht in beiden Fällen genau gleich aus, der Compiler fügt also keinen Spill-Code in die Zwischencode-Darstellung ein (was ja eventuell wieder eine neue Registerbelegung notwendig machen würde). Erst in der Codeerzeugungsphase wird der Spill-Code eingefügt um die Werte vom Speicher zu laden oder dorthin zurück zu schreiben.

5.8.2 Pre-Coloring

Wie sich herausgestellt hat, sind in Java Methodenaufrufe sehr häufig (wie in jeder objektorientierten Programmiersprache). Es ist daher sinnvoll, den Overhead der Aufrufe so klein wie möglich zu halten. Eine wichtige Maßnahme dazu ist es, alle Parameterwerte vor dem Aufruf gleich von Haus aus in die richtigen Register zu schreiben.

Beispiel 9

Source:

```
e = f (a + b, c * d);
```

Zwischencode-Darstellung

```
                                    ; ILOAD 0 (=a)
                                    ; ILOAD 1 (=b)
OP2 (IADD)      I0,I1 -> I5         ; IADD
                                    ; ILOAD 2 (=c)
                                    ; ILOAD 3 (=d)
OP2 (IMUL)      I2,I3 -> I6         ; IMUL
METHOD f        I5,I6 -> I4         ; INVOKESTATIC + ISTORE 4 (=e)
DROP            I5
DROP            I6
```

Es wäre hier sinnvoll, den Pseudoregistern I5 und I6 gleich diejenigen CPU-Register zuzuordnen, die laut den Methodenaufrufskonventionen die Parameterwerte enthalten sollen. Ansonsten wäre es nämlich notwendig, beim Aufruf die Werte extra in die Parameterregister hineinzukopieren (ein zusätzlicher Aufwand der vermieden werden kann und soll).

Diese Pseudoregister kommen im Zwischencode aber schon ein Stück vor dem Methodenaufruf vor, der Registerallokator kann also zum Zeitpunkt, wo er ihnen ein CPU-Register zuordnen will, noch nichts von ihrem Verwendungszweck als Methodenparameter wissen. Aus diesem Grund darf der Compiler schon in der vorherigen Phase (in der der Zwischencode erzeugt wird) für gewisse Pseudoregister eine Art "Empfehlung" abgeben, mit welchem CPU-Register diese zu belegen sind. Er wird also versuchen, für I5 und I6 die CPU-Register par0 und par1 (auf einer Alpha) zu empfehlen. Der Registerallokator versucht dann später natürlich, diese Vorgaben so weit wie möglich einzuhalten, und im Normalfall (wenn nicht gerade verschachtelte Methodenaufrufe oder ähnliches verwendet werden) gelingt es ihm auch.

5.9 Codeerzeugung

Nach all diesen Vorarbeiten ist die tatsächliche Erzeugung von Maschinencode kein Problem mehr. CACAO verzichtet in der derzeitigen Implementierung noch gänzlich auf ein 'Instruction Scheduling' wie es bei gut optimierenden Übersetzern üblich ist [Muc86] oder auf eine optimale Befehlsauswahl [CWF92]. Die erhöhten Kosten der Übersetzung würden sich vor allem bei sehr lange laufenden Programmen (ray tracing, etc.) auszahlen.

Aber zur Zeit geht der Codegenerator den Zwischencode einfach Befehl für Befehl durch und erzeugt die passenden Maschineninstruktionen. Bei Operanden, die in den Speicher ausgelagert sind, muss er vorher eventuell einige Ladeoperationen (zumindest bei Maschinen mit einer Load/Store-Architektur) beziehungsweise nachher eine Speicheroperation einbauen.

Das einzige, was nicht gleich erledigt werden kann, ist die Auflösung der Vorwärtssprünge, aber das kann der Compiler in einer abschließenden Phase mittels Backpatching erledigen.

Eine wichtige Aufgabe der Codeerzeugungsphase ist es auch, den Code für den Anfang und die Beendigung der Methode zu erzeugen. Damit der Code möglichst effizient wird, muss der Codegenerator dabei "leaf"-Methoden erkennen können, damit er bei diesen einen einfacheren Stackbereich anlegen kann (oder in günstigen Fällen gar keinen anlegen braucht). Die Information darüber, ob eine Methode noch weitere Aufrufe enthält, kann schon während der Zwischencode-Erzeugung gewonnen werden. Die Verwendung der Register (und die dadurch entstehende Notwendigkeit, diese Register zu sichern) ergibt sich während der Registerallokation.

5.10 Eingebaute Funktionen

Viele JavaVM-Instruktionen sind zu komplex, als dass sie vom Compiler direkt inline in den Maschinencode eingebaut werden können (zum Beispiel Anweisungen wie: `NEW`, `NEWARRAY`, etc.), statt dessen müssen dafür entsprechende (C-) Funktionen aufgerufen werden. Die Semantik eines solchen Aufrufes entspricht in etwa der einer normalen Java-Methode.

5.11 Übersetzen von JSR/RET

Die JavaVM-Instruktionen `JSR` und `RET` haben den Zweck, Unterprogrammaufrufe **innerhalb** einer Methode durchzuführen. Das ist vor allem dafür sinnvoll, die `finally`-Konstruktion von Java mit möglichst wenig Aufwand auf Bytecode umzuwandeln (allerdings wird dafür die Übersetung auf Maschinencode um so schwieriger).

Der Befehl `JSR` *address*
springt an die angegebene Stelle innerhalb des Bytecodes der Methode und legt

die Rückkehradresse vorher auf den Stack.

Mit `RET` *slot*
kehrt das Programm in die Stelle zurück, die im Variablen-Slot *slot* gespeichert ist.

Aufgrund dieser Asymmetrie muss die Rückkehradresse zuerst vom Stack in einen Slot geschrieben werden (mit `ASTORE`), bevor das `RET` ausgeführt werden kann. Dieses komplizierte Verfahren ist aber die einzige Möglickeit, Parameter am Stack zu übergeben.

Während bei normalen Sprungbefehlen das Ziel des Sprunges zur Zeit der Übersetzung immer gleich festgestellt werden kann, ist das beim JavaVM-Befehl `RET` nicht möglich. Laut Spezifikation [LY96] darf eine Unterroutine (die durch genau ein `RET` abgeschlossen sein muss) von beliebig vielen `JSR` angesprungen werden. Das Hauptproblem bei der Übersetzung ist dabei, die Struktur des Java-Stacks am Ende der Unterroutine wieder an alle Stellen im Programm zurückzumelden, von wo aus sie aufgerufen worden ist (weil diese Information dort für die weitere Bearbeitung notwendig ist).

Zu diesem Zweck merkt sich der Compiler für jedes Unterprogramm (nachdem er den ersten Aufruf übersetzt und das Unterprogramm auch schon durchgearbeitet hat), wie der Stack beim `RET` aussieht, und immer, wenn ein `JSR` auf dieses Unterprogramm erfolgt, macht er gleich mit dieser Stackstruktur weiter.

Am besten kann ich das Ganze wieder einmal an einem Beispiel erklären.

Beispiel für JSR/RET in JavaVM-Code:

```
Block 1, Stackbelegung: leer
     ILOAD   2
     IFNE    Block 3
     ILOAD   0
     JSR     Block 6

Block 2, Stackbelegung: INT
     IRETURN

Block 3, Stackbelegung: leer
     ILOAD   1
     JSR     Block 6

Block 4, Stackbelegung: INT
     JSR     Block 6
```

```
Block 5, Stackbelegung: INT
     IRETURN

Block 6, Stackbelegung: INT, ADDRESS; Endebelegung: INT
     ASTORE  5              ; Dieses seltsame Unterprogramm hat die
     BIPUSH  10             ; Aufgabe, den INT-Wert, der am Stack
     IADD                   ; übergeben wird, um 10 zu erhöhen und dann
     RET     5              ; wieder zurückzukehren
```

Die Reihenfolge der Übersetzung der Blöcke ist hier die folgende:

Block 1: Hier wird die Stackanfangsbelegung für die Blöcke 3 und 6 festgestellt und diese beiden Blöcke gelten ab jetzt als erreicht. Wie der Stack beim Anfang von Block 2 aussehen wird, kann der Compiler jetzt noch nicht sagen. Deshalb wird Block 2 als ein Nachfolgeblock vom Unterprogramm vorgemerkt.

Block 3: Die Anfangsbelegung für Block 6 steht sowieso schon fest (daher wird nichts mehr daran verändert), der Block 4 wird ebenso wie Block 2 als ein Nachfolgeblock des Unterprogramms gemerkt.

Block 6: Block 6 wird fertig abgearbeitet, und die Stackbelegung für das Ende des Unterprogrammes seht ab jetzt fest. Die beiden vorgemerkten Blöcke 2 und 4 bekommen diese Belegung als Anfangsbelegung, und werden als erreicht gekenntzeichnet.

Block 2: Keine weiteren Nachfolgeblöcke

Block 4: In diesem Block wird das Unterprogramm ebenfalls aufgerufen. Die Endebelegung dieses Unterprogrammes steht aber jetzt schon fest, deshalb kann der Compiler die Belegunge gleich für den Block 5 verwenden und mit diesem weitermachen.

Block 5: Keine weiteren Nachfolgeblöcke.

5.12 Andere Zielmaschinen

Ich habe CACAO ursprünglich für die Alpha konzipiert, dabei aber fast alle Teile systemunabhängig gelassen. So ist zum Beispiel die Zwischencode-Darstellung auf allen Maschinen fast gleich, auch die erwähnten Optimierungsverfahren können überall benutzt werden. Die einzigen systemabhängigen Teile sind der Generator für den endgültigen Maschinencode und der Registerallokator.

Unterschiedliche Prozessoren haben aber natürlich einen unterschiedlichen Befehlsumfang, und können nicht alle JavaVM-Instruktionen in gleichem Umfang unterstützen. Ein gutes Beispiel ist die IDIV und IREM - Instruktion der JavaVM. Bei den meisten Prozessoren kann diese Operation direkt in den Code hineincompiliert werden, die Alpha beispielsweise hat aber keine eingebaute Division, deshalb braucht der Compiler statt dessen einen Unterprogrammaufruf. In solchen Fällen unterscheidet sich dann auch die Zwischencode-Darstellung bei unterschiedlichen Maschinen (einmal ein Aufruf eines Unterprogrammes, das andere mal steht dort nur eine `OP2 (IDIV)`-Instruktion).

Aber nicht nur die Unterschiede in den Instruktionssätzen ist hier wichtig, sondern auch die Art und Menge der verfügbaren Register. In der Registerallokationsphase werden die (ziemlich abstrakt definierten) Pseudoregister auf tatsächliche CPU-Register abgebildet. Auf der Alpha geht das recht einfach, weil diese CPU volle 64 Bit sowohl für Integer- als auch für Fließkommawerte unterstützt. Die Java-Typen `LONG` und `DOUBLE` sind hier also ohne Probleme unmittelbar darstellbar. Auf Maschinen mit 32-Bit-Registern (Maschinen mit noch kürzeren Registern ziehe ich gar nicht mehr in Betracht) braucht man für solche Werte entweder ein Registerpaar, oder gleich einen Speicherplatz am Stack selbst (das Auslagern auf den Speicher muss sowieso jede Implementierung unterstützen, weil es bei sehr großen Java-Methoden zwangsläufig dazu kommen wird).

Auf zusätzliche Unterschiede wie Aufrufskonventionen oder Adressierungsarten möchte ich hier aber nicht näher eingehen.

Kapitel 6

Die Exceptions

Bisher habe ich die Schwierigkeiten, die sich bei der korrekten Behandlung von Java-Exceptions ergeben, nicht erwähnt um die Beispiele nicht zu sehr zu komplizieren. Tatsächlich bildet die Ausnahmebehandlung aber einen integralen Bestandteil jeder (auch dieser) JavaVM-Implementierung. Deshalb hier das Wichtigste in Kürze:

Ich setze für die folgende Erklärung voraus, dass die Konzepte der Exceptions und Exception-Handler von Java bekannt sind [LY96].

Damit das Programm zur Laufzeit alle Ausnahmesituationen korrekt erkennen kann, muss der Compiler notgedrungen Laufzeit-Überprüfungen in den Maschinencode einbauen. So zum Beispiel bei allen Integer-Divisionen oder Array-Zugriffen. Diese Tests sind unter Umständen recht aufwendig (Werte aus dem Speicher lesen, Test auf 0 durchführen, etc.) und können den Programmlauf ziemlich bremsen. Aus diesem Grund sind manche dieser Tests beim Aufruf von CACAO abschaltbar (ist aber nur bei wirklich verlässlichen Programmen zu empfehlen).

Die normale Ausnahmebehandlung geht aber so vor sich: Ein entsprechender Laufzeit-Test wird durchgeführt, und wenn der Test eine Ausnahmesituation erkennt, dann wird ein Programmstück (das irgendwo weit abseits im Maschinencode plaziert ist) aufgerufen, das dann folgende Dinge tun muss:

- Ein Exception-Objekt vom passenden Typ erzeugen (beispielsweise eine `ArrayIndexOutOfBoundsException`).
- Den passenden Exception-Handler anspringen (Jede JavaVM-Instruktion kann ja von solchen Exception-Handlern "bewacht" werden) beziehungsweise eine außerordentliche Methoden-Terminierung auslösen (wenn kein passender Handler vorhanden ist).

Jede Methode kann möglicherweise mit einer Exception terminieren, und diese muss dann in der aufrufenden Methode weiter behandet werden. Dazu braucht diese aber die Information darüber, ob eine und welche Exception vorliegt.

Zu diesem Zweck habe ich für alle Methodenaufrufe einen zusätzlichen Rückgabewert vorgesehen, der den Zeiger auf ein allfälliges Exception-Objekt enthält. Im Normalfall (gewöhnliche Methodenbeendigung) enthält dieser Zeiger `NULL` und das Programm soll normal weiterlaufen. Bei einer Ausnahmebehandlung (Zeiger ungleich `NULL`) muss sofort der entsprechende Exception-Handler angesprungen beziehungsweise auch die aufrufende Methode sofort terminiert werden.

Dieser NULL-Test kann (zumindest auf der Alpha) mit einer einzigen `BNE`-Verzweigung durchgeführt werden, der Aufwand ist also nur ein Maschinenzyklus pro Methodenaufruf (vor allem deshalb, weil so eine Verzweigung in 100 Prozent der Fälle als "predict not taken" vom Prozessor vorhergesagt werden kann [HP96]). Ich halte diese Variante deshalb für sinnvoller als das Verfahren, jedesmal, wenn ein `try`-Block anfängt, den gesammten Prozessorzustand in einer Datenstruktur abzuspeichern und im Fall einer Exception direkt am innersten `catch` wieder aufzusetzten (das könnte sofort zu gigantischem Aufwand ausarten, wenn das Programm nämlich permanent in `try`-Blöcke hinein- und wieder herausläuft).

Bei einer solchermaßen weitergeleiteten und erkannten Exception kann der zu verwendende Handler (von denen es für eine JavaVM-Instruktion ja mehrere geben kann, und die sich durch den Typ der abgefangenen Exception unterscheiden) zur Übersetzungszeit nicht feststehen, und so muss der Compiler Laufzeit-Typchecks einbauen um den tatsächlichen Exception-Handler zu bestimmen.

Auch hier kann ich hoffentlich die Sache mit einem Beispiel erklären:

Exception-Beispiel: Java-Sourcecode

```
try {
     b = c/d;
     a = dosomething (b);
}
catch (ArrayIndexOutOfBoundsException e) {
     a = 0;
}
```

Exceptions-Beispiel: JavaVM-Code

```
l1:  ILOAD          c
     ILOAD          d
     IDIV
     ISTORE         b
     ILOAD          b
     INVOKESTATIC   dosomething (INT) INT
     ISTORE a
l2:  GOTO           l4
l3:  POP
     ICONST         0
     ISTORE         a
l4:  ..

Exceptions-Tabelle:
     von l1 bis l2 bei ArrayIndexOutOfBoundsException -> l3
```

Exceptions-Beispiel: Zwischencode

```
Block 1, Anfangs-Stack: leer
                                                 ; ILOAD c (=I2)
                                                 ; ILOAD d (=I3)
     BRA (IFEQ)          I2, Block 10000         ; (IDIV-Check)
     METHOD (builtin)    idiv / I2,I3 -> I1,-    ; IDIV + ISTORE b (=I1)
                                                 ; ILOAD b (=I1)
     METHOD (STATIC)     doit / I1 -> I10,A4     ; INVOKESTATIC
     BRA (IFNONNULL)     A4, Block 10001
     MOVE (INT)          I10 -> I0               ; ISTORE a (=I0)
     DROP                I10
     BRA (GOTO) ..                               ; GOTO ...

Block 2, Anfangs-Stack: A4
     DROP                A4                      ; POP
     LOADCONST (INT)     #0 -> I0                ; ICONST_0 + STORE a (=I0)

...

Block 10000, Anfangs-Stack: leer
     LOADCONST (ADR)     &ArithmeticException_Prototype -> A5
     IRETURN -,A5
```

```
Block 10001, Anfangs-Stack: leer; Exeception-Zeiger in: A4
     LOADCONST (ADR)     &ArrayIndexOutOfBoundsException -> A7
     METHOD (builtin)    isinstanceof / A4,A7 -> I8,-
     BRA (IFNE)          I8, Block 2
     IRETURN -,A4
```

Die Blöcke 10000 und 10001 (die intern erzeugt wurden, ohne dass ihnen eine entsprechende JavaVM-Codesequenz gegenübersteht) enthalten den Code, der im Falle von Ausnahmebehandlungen aufgerufen wird.

Der Block 10000 wird gestartet, wenn die IDIV - Instruktion als Divisor eine 0 erhalten würde. In diesem Fall erzeugt das Programm eine `ArithmeticException` (oder besser gesagt, es verwendet den Zeiger auf ein schon im vorhinein bereitgelegtes Objekt) und terminiert die Methode und gibt den Zeiger auf das Objekt als weiter zu behandelnde Exception zurück. Es gäbe zwar in diesem JavaVM-Code einen Exception-Handler, der die IDIV-Instruktion bewacht, aber der Typ der abgefangenen Exception passt nicht auf die `ArithmeticException`.

Der Block 10001 wird gestartet, wenn der Methodenaufruf (der `static` Methode "doit") schon eine Exception "geworfen" hat. Der zweite Rückgabewert (im Pseudoregister A4) enthält gegebenenfalls einen Zeiger auf das Exception-Objekt. An dieser Stelle muss das Programm zuerst feststellen, ob das Objekt von der Klasse `ArrayIndexOutOfBoundsException` (oder einer abgeleiteten Klasse) ist. Das wird mit dem Aufruf der eingebauten Funktion `isinstanceof` gemacht, und wenn das Objekt passt, dann wird der dafür vorgesehene Exception-Handler (Block 2) aufgerufen. Im anderen Fall terminiert die Methode.

Die eingebauten Funktionen (zum Beispiel `isinstanceof` oder `idiv`) lösen niemals eine Exception aus (`idiv` etwa würde bei ungültigem Dividenden einfach abstürzen). Der Compiler muss also den Code so erzeugen, dass die Parameter auf jeden Fall immer gültig sind, und dass er den Rückgabewert richtig auswertet (beispielsweise bedeutet der Rückgabewert `NULL` bei der eingebauten Funktion `NEW`, dass der Speicher aus ist, und die entsprechende Exception geworfen werden muss).

Bei dieser Art, die geworfenen Exceptions an übergeordnete Methoden weiterzureichen, ist auf jeden Fall ein zusätzlicher Verzweigungsbefehl pro Methodenaufruf notwendig (der Compiler kann auf keine Weise im vorhinein feststellen, ob eine Methode eine Exception erzeugen wird oder nicht).

Es gäbe auch eine ganz andere Art für die Behandlung der Exceptions: Nur bei den Methoden, die tatsächlich Exception-Handler besitzen, werden in einer zusätzlichen Datenstruktur (möglicherweise auch auf dem Aufrufstack) alle notwendigen Informationen (Registerbelegung, Stackgröße, etc.) gespeichert, um im Falle einer Ausnahmebehandlung gleich direkt den innersten gültigen Exception-

Handler anzuspringen. Dieses Verfahren ist sicherlich um einiges komplizierter, wenn es auch bei Programmen, die sehr selten in einen `try`-Block hineinlaufen, schneller wäre. Nur bei Programmen, die immer wieder `try`-Blöcke betreten und wieder verlassen, müsste jedesmal die notwendige Datenstruktur umgebaut werden, was jede Menge Speicherzugriffe und großen Zeitaufwand bedeutet.

Aber wie gesagt: Ich wollte das System so einfach wie möglich halten, deshalb verwende ich bei CACAO eben die erste Variante.

Kapitel 7

Übersetzung "Just in Time"

Es gäbe prinzipiell zwei Möglichkeiten dafür, zu welchem Zeitpunkt die Übersetzung einer Methode vom Bytecode auf Maschinencode erfolgen soll, nämlich zum Zeitpunkt des Ladens des Klassenfiles oder aber erst dann, wenn die Methode tatsächlich gebraucht (also aufgerufen) wird. Für die erste Variante spricht, dass der Overhead (fürs Starten des Compilers und fürs Abfangen der Aufrufe noch nicht übersetzter Methoden) ein wenig kleiner ist. Die zweite Variante hat den Vorteil, dass gar nicht alle Methoden, die geladen wurden, auch jemals benötigt werden, und hier deshalb auch nicht übersetzt werden müssen (Das gilt vor allem für die vielen, vielen Bibliotheks-Methoden).

Ich habe mich daher eben für die zweite Variante entschieden und CACAO übersetzt die Methoden erst zum Zeitpunkt ihres ersten Aufrufes.

Zu diesem Zweck werden in allen Methodentabellen (Virtual Function Table, Methodentabellen der Interfaces, etc.) beim Start von CACAO nur Zeiger auf sogenannte Stub-Routinen eingetragen, deren einzige Aufgabe es ist, gegebenenfalls den Compiler zu starten. Jede dieser Stubs dient zur Übersetzung nur einer einzigen Methode und sollte deshalb sehr kurz sein um nicht zuviel Speicher zu vergeuden.

Auf der Alpha sieht so einen Stub folgendermaßen aus:

```
start:        ; Register pv zeigt beim Aufruf hierher
     LDQ      r0,(p1-start)(pv)   ; Zeiger auf Bytecode in R0 laden
     LDQ      pv,(p2-start)(pv)   ; Zeiger auf den Compiler laden
     JMP      zero,(pv)           ; Compiler anspringen
p1:  .QUAD    methodinfo
p2:  .QUAD    jit_compiler
```

Ein ganzer Stub dieser Art hätte also ein Länge von 32 Byte.

Ein zweites Detailproblem der Just-In-Time - Übersetzung besteht darin, alle Methodenzeiger in den verschiedenen Tabellen (und es gibt da eine ganze Menge von verschiedenen Orten, wo diese Zeiger gespeichert werden) auf den neuen Stand zu bringen.
Auch hier gibt es wieder zwei Möglichkeiten:

- Erste Variante: Der Compiler sucht, nachdem er eine Methode übersetzt hat, alle Tabellen durch und ersetzt alle Zeiger, die noch auf den Java-Bytecode zeigen, durch Zeiger auf den gerade erzeugten Maschinencode.
- Zweite (von mir verwendete) Variante: Der Compiler ersetzt nur diejenigen Zeiger, die tatsächlich gebraucht werden (die Just-In-Time - Idee in letzter Konsequenz).

Um diese zweite Möglichkeit zu verwirklichen, muss der Compiler aber zur Laufzeit herausfinden, welche Einträge aus welchen Methodentabellen tatsächlich benutzt werden.
Das macht er (zumindest jetzt in der Alpha - Implementierung) auf folgende Weise:

- Alle Methodenaufrufe, die der Compiler erzeugt, haben den selben Aufbau:

```
...
LDQ  pv,offset(R28)    ; Zeiger auf die Methode laden
JMP  ra,(pv)           ; Methode aufrufen
```

- Mit Hilfe der Rücksprungadresse im Register `ra` kann der Compiler genau diejenige Stelle im Maschinencode finden, von der aus die Methode aufgerufen wurde.
- Indem der Compiler aus der oben beschriebenen `LDQ` - Maschineninstruktion (die an der Adresse `ra` minus 8 zu finden ist) den Wert von `offset` extrahiert und zum Register `R28` addiert, kann er herausfinden, von wo der Methodenzeiger ursprünglich geladen wurde.
- Jetzt ersetzt der Compiler diesen Zeiger (der bislang nur auf die Stub-Routine gezeigt hat) durch einen Zeiger auf den tatsächlichen Maschinencode.
- Alle späteren Aufrufe der Methode, die mit Hilfe des selben Zeigers passieren, erfolgen dann ohne weitere Verzögerung.

Kapitel 8

Das Laufzeitsystem

8.1 Laden, Linken und Initialisieren der Klassen

Wenn CACAO eine Klasse laden muss (entweder die eine Klasse, die beim Systemstart angegeben wurde, oder später eine dynamisch referenzierte Klasse), dann lädt er auch gleich *alle* von dort aus referenzierten Klassen, so dass danach auf jeden Falle alle Referenzen aufgelöst sind.

Nachdem alle irgendwo notwendigen Klassen in den Speicher geladen wurden, beginnt die zweite Phase (Linking). Hier werden die oben erwähnten Methodentabellen aufgebaut und die Stub-Routinen (für die nachfolgende Just-In-Time - Übersetzung) erzeugt.

Initialisieren einer Klasse bedeutet (laut JavaVM-Specification), die `static` Methode `<clinit>` einmal aufzurufen, und zwar unmittelbar vor der ersten "active use" einer Klasse.

"Active use" ist (ebenfalls laut JavaVM-Spec) entweder ein Aufruf einer Methode der Klasse, oder ein Zugriff auf eine ihrer Klassenvariablen.

Für CACAO ist es kein Problem, herauszufinden, wann eine Methode das erste mal aufgerufen wird, nämlich genau dann, wenn auch die Übersetzung dieser Methode initiert wird (Methoden werden erst zum letztmöglichen Zeitpunkt übersetzt).

Schwieriger ist es, herauszufinden, wann genau das (übersetzte und in Maschinencode ablaufende) Programm auf irgendwelche Klassenvariablen zugreift. Deshalb betrachtet CACAO eine Klassenvariable als benutzt, sobald der Compiler eine Methode auch nur übersetzt, die auf die Variable zugreifen könnte (auch wenn dieser Zugriff letztlich nicht passiert, weil er irgendwo in einer bedingten Anweisung steckt).

Diese kleine Inkompatibilität zur JavaVM-Specification sollte sich (hoffentlich!) in keiner vernünftig programmierten Anwendung negativ auswirken.

8.2 `native` Methoden

Die Zugriffe auf das Betriebssystem (und dafür werden die `native` Methoden eigentlich verwendet) werden sinnvollerweise in einer Sprache wie "C" programmiert. In CACAO (das ich ja zur Gänze in "C" entwickelt habe) sind sie es auch tatsächlich.

Leider können diese Funktionen nicht einfach als Java-Methoden aufgerufen werden, weil sie ein wichtiges Konzept nicht unterstützten, nämlich die außerordentlich Terminierung mit einer Exception. Aus diesem Grund braucht jede `native` Methode (ebenso wie jede noch nicht übersetzte Methode) eine kleine Stub-Routine, die Aufrufskonventionen von Java auf diejenigen von "C" umsetzt. Dazu ist in der Alpha-Version von CACAO nichts anderes nötig, als nach Beendigung der Methode den Zeiger auf die zu werfende Exception aus einer globalen Variablen in ein Register zu laden (das ist nämlich der ganze Unterschied zwischen den Aufrufskonventionen). Der Stub ist also relativ kurz.

Diese `native` Methoden müssen sich aber an eine wichtige Einschränkungen halten, damit das ganze System nicht zusammenbricht:

Sie dürfen keine Zeiger auf Objekte so aufbewahren, dass sie der Garbage Collector nicht finden kann.

Wenn sie es dennoch unbedingt tun müssen, dann können sie dem GC explizit die Speicherstellen mitteilen, an denen Referenzen gespeichert sind.

8.3 Multi-Threading

Multi-Threading habe ich in der derzeitigen Version von CACAO noch nicht implementiert.

Das Hauptproblem dabei wird sein, einen eventuellen Stacküberlauf in den Threads zu verhindern. Bei der jetzigen Implementierung verwendet CACAO nämlich einfach den System-Stack, der zur Laufzeit ohne Beschränkung wachsen darf.

Eine Möglichkeit wäre, in einem zusätzlichen Register die Grenze des Stacks mitzuführen, und bei jedem Methodenaufruf zu überprüfen, ob diese Grenze überschritten werden würde. In diesem Fällen muss die Methode eben einen `StackOverflowError` generieren.

8.4 Speicherbereinigung

Die Java Virtual Machine sieht eine automatische Freigabe (Garbage Collection) aller nicht mehr benötigter Objekte vor. Wie diese Freigabe genau passieren soll, ist aber nicht näher spezifiziert, sondern bleibt ganz dem Einfallsreichtum der

Entwickler eines Java-Laufzeitsystems überlassen.

Eine konservative Definition dafür, welche Objekte später noch benötigt werden könnten, und welche nicht, ist diese:
Alle Objekte, die ein JavaVM-Programm noch irgendwie (auch über Umwegen) durch gültige Zeigeroperationen erreichen kann, müssen als noch benötigt angenommen werden. Alle anderen Objekte sind nur mehr "Garbage" und können gelöscht werden.

In der Literatur (wie zum Beispiel in [JL96]) gibt es sicher -zig verschiedene Verfahren, wie nicht mehr benötigte Speicherbereiche erkannt und freigeben werden können. Die einen sind schneller, dafür brauchen sie auch mehr Speicher, andere benötigen intensive Unterstützung durch das laufende Programm, wieder andere können parallel zum übrigen Programm arbeiten, oder sogar auf verteilten Maschinen.

Wichtige Kriterien, wo und wie diese Verfahren eingesetzt werden können, sind unter anderem, ob und für wie lange der Programmablauf für die Speicherbereinigung unterbrochen werden darf, die Menge an zusätzlich notwendigem Speicher und der gesammte Rechenaufwand für den Garbage Collector.

Ich will hier nur ein paar der wichtigsten Methoden (die sich darüberhinaus natürlich auf die verschiedensten Arten zu immer raffinierteren Systemen kombinieren lassen) erwähnen, und deren Vor- und Nachteile kurz erläutern.

8.4.1 Reference Counting

Reference Counting ist wahrscheinlich der anschaulichste und in gewisser Weise einfachste Algorithmus. Dazu wird bei jedem Objekt ein Zähler gespeichert, der angibt, von wie vielen Stellen aus auf das Objekt verwiesen wird. Sobald der Zähler auf 0 sinkt, darf das Objekt gelöscht werden (und bei allen Objekten, auf die von dem Objekt referenziert wird, wird deren Zähler um 1 kleiner)

Vorteile

Speicher wird sobald wie möglich freigegeben.

Das Verfahren muss den Programmablauf nicht unterbrechen.

Nachteile

Relativ großer atomarer Aufwand für jede Zeiger-lade- bzw. -speicheroperation.

Zusätzliches Zählerfeld für jedes Objekt.

Zyklische Referenzen werden nicht freigegeben.

8.4.2 Mark and Sweep

Mark-and-Sweep beruht auf der obigen formalen Definition der Erreichbarkeit von Objekten. Ausgehend von einer bekannten Menge von Referenzen (globale und lokale Variablen) werden alle irgendwie von dort (möglicherweise über große Umwege) aus erreichbaren Objekte markiert. Diese Markierungsphase ist auf einfache Weise nur als eine atomare Operation möglich, während der der

Programmablauf angehalten werden muss. Die zweite Phase (die Sweep-Phase) entfernt alle nicht markierten Objekte und gibt deren Speicher frei.

Vorteile Kein zusätzlicher Aufwand während der normalen Verarbeitung.

Auch zyklische Referenzen werden freigegeben.

Nachteile Großer Aufwand während der Markierungsphase, während der das Programm angehalten werden muss.

8.4.3 Scavenging / Copying

Dieses Verfahren soll den großen Aufwand, der bei Mark-and-Sweep fürs entfernen der nicht mehr erreichbaren Objekte anfällt, eliminieren. Dazu werden alle Objekte, die während der Markierungsphase erreicht werden, in einen ganz anderen Speicherbereich kopiert, und befüllen diesen Speicherbereich von unten nach oben (So wird zusätzlich die Fragmentierung des Speichers beseitigt). Der Originalspeicher wird als Ganzes freigegeben, und kann beim nächsten GC als Zielbereich dienen. Zusätzlicher Bonus: Der freie Speicher bildet einen einzigen Block, das Anlegen eines neuen Objektes ist daher sehr einfach (durch Erhöhen eines Index möglich).

Vorteile Der Aufwand für die Operation ist proportional zur Menge der lebenden Objekte.

Anlegen eines neuen Objektes ist sehr einfach.

Keine Speicherfragmentierung

Nachteile Großer Kopieraufwand.

Alle Zeiger auf die Objekte müssen umgesetzt werden.

Zusätzlicher Speicher für Zielbereich notwendig.

8.4.4 Generations

Man kann in real existierenden Progammen beobachten, dass Objekte entweder sehr lange leben (irgendwelche statischen Informationen), oder aber ziemlich bald nach ihrer Erzeugung wieder weggeworfen werden [Ung84] [LH83]. (Ein Beispiel wären Objekte, die Eingabeaktionen des Benutzers repräsentieren)

Weil man annehmen kann, dass Objekte, die schon eine gewisse Zeit lang überlebt haben, unbeschadet noch eine Weile aufgehoben werden können, teilt man die Menge aller Objekte in verschiedene Generationen ein. Bei jeder Generation führt man das Garbage Collection in unterschiedlichen Intervallen durch, je älter die Generation, umso seltener. Problematisch wird es, wenn Referenzen zwischen Objekten verschiedener Generationen existieren, insbesondere bei einem Zeiger von einem alten Objekt auf ein neueres. Um diese Dinge zu erkennen, muss das laufende Programm irgendwie Informationen über das Alter der Objekte haben, und wenn es eine Referenz in ein Objekt speichert, überprüfen, ob noch alles konsistent ist.

Vorteile Weniger Aufwand für Mark-and-Sweep-Durchgang

Kürzere Pausen im Programmlauf

Nachteile Komplizierte Ausnahmebehandlungen während der Laufzeit

8.4.5 Speicherbereinigung für meine JavaVM - Implementierung

Der Garbage Collector ist einer der letzten Teile, die ich in das System integriert habe, und musste deshalb eine Menge Vorgaben durch das übrige System erfüllen:

- Die lebenden Objekte dürfen im Speicher nicht verschoben werden.
- Es dürfen keine zusätzlichen Aktionen während des Programmlaufes erforderlich sein.
- Es gibt keine zusätzliche Information über den Inhalt des Stacks.
- Alle Aktionen des Garbage Collectors müssen während des Anlegens neuer Objekte passieren.
- Es soll so wenig zusätzlicher Speicher benötigt werden wie möglich.

Dafür ist es für normale Einsatzzwecke von Java aber nicht notwendig, bei jedem Anlegen eines neuen Objektes eine garantierte Antwortzeit einzuhalten (Inkrementelle Algorithmen sind also nicht notwendig).

Allein durch diese Vorgaben fallen schon fast alle raffinierteren Methoden weg:

- Alle Arten von Scavenging und Copying, weil die Objekte nicht verschoben werden dürfen.
- Reference Counting, weil man dazu während der Programmlaufzeit permanent die Zähler modifizieren müsste.
- Alle Verfahren, die die Objekte in Generationen unterteilen, weil man dort zur Laufzeit zusätzliche Überprüfungen durchführen muss, nämlich, zum Beispiel, ob in ein altes Objekt ein Zeiger auf ein neueres Objekt eingetragen wird. (Mit entsprechender Hardware-Unterstützung durch die MMU liese sich das zwar ohne Leisungseinbußen machen, solche Spezialitäten des Betriebssystems sind aber auf jedem Rechner anders und CACAO wäre dann noch weniger portabel).

Was dann noch übrig bleibt, ist nur mehr ein einfaches Mark-and-Sweep-Verfahren, noch dazu mit der Einschränkung, dass der Gargabe Collector nur sehr wenig Information darüber hat, ob und wo genau Zeiger auf Objekte im Heap gespeichert sind. Er muss sich also stellenweise auf ein "Conservative Pointer Finding" beschränken, das heisst, bei allem was irgendwie ein Zeiger in den Heap sein könnte, muss er davon ausgehen, dass es tatsächlich einer ist, und darf das dort liegende Objekt nicht freigeben.

Aus der Unsicherheit, ob ein Zeiger jetzt tatsächlich auf ein gültiges Objekt am Heap zeigt, folgt unmittelbar, dass der Garbage Collector eine zusätzliche Datenstruktur braucht, mit der er sich darüber Sicherheit verschaffen kann.

Ich habe drei recht ähnliche Arten ausgearbeitet, wie man die zusätzlich notwendige Information in kompakter Form speichern kann. Dabei wird der Heap in lauter gleich große Blöcke aufgeteilt (typischerweise 8 Byte), die immer als eine Einheit vergeben werden. Für jeden dieser Blöcke gibt es dann zusätzliche Information über dessen Verwendung.

8.4.6 1. Variante: 1 Bit pro Block plus Zusatzinfos im Objekt

Für jeden Speicherblock im Heap wird ein Bit in einem Bitfeld reserviert, das anzeigt, ob in diesem Block gerade ein neues Objekt anfängt (damit kann sich der Garbage Collector Sicherheit darüber verschaffen, ob ein vermeintlicher Zeiger nun wirklich auf ein gültiges Objekt zeigt).

Diese Datenstruktur allein reicht aber noch nicht aus. Vor allem ist noch Speicher für die Markierungsbits der Sweep-Phase notwendig, beziehungsweise irgendeine Längeninformation für die einzelnen Objekte.

Zu diesem Zweck kann man jedes Objekt noch um zusätzliche Datenfelder erweitern, nämlich:

- die Länge das Objektes (in Byte)
- ein Markierungsbit
- ein Kennzeichnungsbit, ob das Objekt Referenzen enthält

Um Platz zu sparen, kann man aber die zusätzlichen Informationsbits als niederwertigste Bits im Längenfeld abspeichern (weil die Länge sowieso immer ein Vielfaches der Blockgröße ist).

Alle Frei-Speicher-Bereiche können ebenfalls alle diese Felder enthalten, damit wird dann die Sweep-Phase sehr einheitlich. Der Garbage Collector muss sich nur mehr von Objekt zu Objekt durch den Heap durcharbeiten und dabei alle freien Bereiche (also alle Bereiche, deren Markierungsbit gelöscht ist) zu einer Freispeicherliste zusammensetzen. Bei allen belegten Bereichen kann er gleich die Markierungsbits (für die nächste Markierungsphase) löschen.

8.4.7 2. Variante: 2 Bit pro Speicherblock

Diese Variante hat den kleinsten Speicherverbrauch. Hier werden für jeden Block jeweils 2 Bit in zwei fortlaufenden Bitfeldern reserviert (also ein Bit in jedem Feld). Diese beiden Bits pro Block (ich nenne sie S (=Start) und M (=Markierung)) haben folgende Bedeutung:

Das S-Bit ist genau bei den Blöcken gesetzt, wo ein Objekt oder ein freier Speicherbereich (!) anfängt, die M-Bits zeigen an, dass das Objekt noch irgendwie erreichbar ist. In der Markierungsphase werden alle M-Bits zuerst gelöscht, und dann alle Objekte, die noch vom Programm aus erreichtbar sind, markiert. Dazu folgt der Markierer allen möglichen Referenzen, und entscheidet mit Hilfe des S-Bits, ob der Zeiger auf ein gültiges Objekt zeigt. Wenn ja, dann wird das Objekt markiert (das M-Bit wird gesetzt, wenn es noch nicht gesetzt war), und die gesammten Daten im Objekt werden gleich wieder als potentielle Zeiger angenommen und weiter behandelt. Das Ende des Objektes wird durch das nächste folgende S-Bit markiert (deshalb müssen auch alle Freispeicherbereiche so ein S-Bit am Anfang haben).

Es kann hier (aber hoffentlich selten) passieren, dass ein freier Speicherbereich fälschlicherweise als gültiges Objekt erkannt und markiert wird. Dieser Speicher geht dem Garbage Collector zwar fürs Erste verloren, in späteren Säuberungsdurchläufen wird der Speicher aber hoffentlich wieder freigegeben (diese Unsicherheit entsteht hauptsächlich durch das "Conservative Pointer Finding", sie wirkt sich aber im tatsächlichen Einsatz eigentlich nicht merkbar aus).

Nach der Markierungsphase muss der GC die Freispeicherliste der unbenutzten Bereiche anlegen. Jeder jetzt freie Bereich ist nach wie vor mit einem S-Bit gekennzeichnet, aber das M-Bit ist nicht gesetzt (S AND NOT M). Das Ende eines freien Bereiches erkennt der GC daran, dass sowohl das S-Bit als auch das M-Bit gesetzt sind (S AND M). Mit Hilfe von logischen Operatoren und einer schnellen Methode, die Stelle des ersten gesetzten Bits in einer Integer-Zahl zu finden, lässt sich diese Arbeit recht rasch erledigen (Es müssen ja nur die jetzt *freien* Bereiche einzeln behandelt werden, die Anzahl der lebenden Objekte ist hier ziemlich irrelevant). Besonders praktisch ist dabei auch, dass nebeneinanderliegende Freispeicherbereiche ganz automatisch zusammengefasst werden.

Das einzige noch verbleibende Problem ist das, alle S-Bits (auch diejenigen der freien Bereiche) richtig zu setzen. Beim automatischen Zusammenlegen freier Bereiche werden diese Bits ja nicht verändert. Eine Lösung ist, bei jedem Anfang eines neu erkannten freien Bereiches (S AND NOT M) nachträglich das M-Bit doch zu setzen. Ganz am Schluss werden dann ganz einfach alle M-Bits über die S-Bits kopiert (oder was noch schneller geht, die beiden Bitfelder werden einfach vertauscht). Der Effekt ist dann: Alle lebenden Objekte haben ein S-Bit gesetzt (weil sie markiert worden sind), und alle freien Speicherbereiche ebenfalls (weil nachträglich die M-Bits eingefügt worden sind).

8.4.8 3. Variante: 3 Bit pro Speicherblock

Der gravierendste Nachteil der vorigen Variante ist der, dass der GC bei jedem gefundenen Objekt auf jeden Fall alle darin enthaltenen Daten als mögliche Zeiger auffassen muss. Er muss diese Daten also immer aus dem Speicher laden und überprüfen, was erstens viele Instruktionen und zweitens viele Speicherzugriffe (mit jeder Menge Cache-Misses) benötigt.

Tatsächlich enthalten aber viele Objekte überhaupt keine Zeiger auf andere Objekte mehr (Character-Arrays, etc.), und hier könnte der GC mit ein wenig Hilfe viel Zeit sparen.

Eine Lösung ist, jedem Speicherblock noch ein drittes R-Bit (=Referenzen) zu spendieren, das anzeigt, ob das dort anfangende Objekt überhaupt irgendwelche Referenzen enthalten kann. Diese Information steht schon fest, sobald das Objekt angelegt wird, und verursacht zu diesem Zeitpunkt einen sehr kleinen einmaligen Aufwand. (Noch besser wäre natürlich, gleich für jeden Block innerhalb des Objektes ein separates R-Bit einzutragen. In der derzeitigen Version führt der Compiler aber keine expliziten Tabellen darüber mit, wo genau in den Objekten Zeiger stehen können).

Der Speicherverbrauch ist ebenfalls recht moderat, aber die Zeitersparnis ist doch ganz wesentlich, vor allem, weil der GC bei Objekten, die eben keine Referenzen enthalten, den Inhalt der Objekte niemals aus dem Speicher lesen muss, sondern immer mit den drei Bitfeldern auskommt (die, weil sie so kompakt gespeichert sind, den Cache recht gut ausnützen werden).

Ich habe mich für CACAO deshalb für diese dritte Variante entschieden.

Kapitel 9

Benchmarks

9.1 Applikationen

Ich habe CACAO mit zwei anderen Java-Bytecode-Compilern verglichen:

- Guava ist ebenfalls so wie CACAO ein Just-In-Time Compiler, der die Klassenfiles erst zu Laufzeit in Maschinencode umsetzt.
- Toba hingegen ist ein System, das die Klassenfiles in C-Programme übersetzt, die dann mit einem C-Compiler weiter bearbeitet werden müssen. Die Benchmarks berücksichtigen den Übersetzungsaufwand des Toba nicht, sondern nur mehr die reine Laufzeit (was den Vergleich natürlich ziemlich zugunsten dieses Systems verzerrt).

Leider laufen diese beiden Übersetzer unter Solars auf einer SparcStation (eventuell noch unter Linux), CACAO aber ausschließlich auf Rechnern mit einer Alpha-CPU. Ich habe deshalb keinen direkten Geschwindigkeitsvergleich durchführen können. Die Benchmarks setzen die Leistung der Compiler in Verhältnis zur Leistung des Bytecode-Interpreters aus dem "Java Development Kit" (JDK). Auffallend ist dabei, dass CACAO in allen Fällen schneller ist als Guava und sogar auch oft schneller als Toba (obwohl bei Toba die Übersetzungszeit nicht in die Laufzeit mit eingerechnet wird).

Die Laufzeitdaten für Toba und Guava stammen von der Toba-Entwicklungsgruppe selbst aus deren WWW-Seite:

`http://www.cs.arizona.edu/sumatra/toba/benchmarks.html`

Die Daten für CACAO habe ich mit genau den selben Java-Programmen und den selben Testfällen erzeugt.

`JavaLex` ist ein Scanner-Generator, der aus einer Grammatik (in diesem Test aus der mitgelieferten 'tiger.lex'-Grammatik) einen Scanner in Form eines Java-Sourceprogrammes erzeugt.

`javac` ist der Compiler aus dem Java Development Kit (JDK) und wird in dem Test verwendet, um die Sourcen von Toba zu übersetzen.

`espresso` ist die alte Version von `pizza`, eines anderen Java-Compilers. In diesem Test wird er ebenfalls verwendet, um die Sourcen von Toba zu übersetzen.

`Toba` ist ein Java-to-C - Compiler, der direkt aus Java-Bytecode C-Programme erzeugt. Toba wird hier verwendet, um seine eigenen Klassenfiles zu übersetzen.

`java_cup` ist ein Parser-Generator, der aus einer Grammatik einen Parser als Java-Source erzeugt. In diesem Test generiert er den Sourcecode für sich selbst (aus der Definitionsdatei `parser.cup`, die bei der Distribution von `java_cup` dabei ist).

	JavaLex	javac	espresso	Toba	java_cup
Laufzeiten auf einer SparcStation 20 (in Sekunden)					
JDK	176.3	45.0	24.9	59.8	8.3
GUAVA	80.8	17.8	-	36.8	6.2
TOBA	38.8	12.8	4.9	20.1	2.9
Laufzeiten auf einer 21064A 300MHz (in Sekunden)					
JDK	29.8	18.5	8.7	32.1	3.5
CACAO total	2.65	4.74	3.17	4.58	1.52
load	0.18	0.61	0.38	0.18	0.21
compile	0.39	1.21	0.91	0.55	0.81
run	2.08	2.92	1.88	3.85	0.50
Anzahl der von CACAO übersetzten JavaVM-Befehle					
	13412	34759	27281	14430	17489
Speedup im Verhältnis zum Interpreter					
speedup JDK/GUAVA	2.18	2.53	-	1.62	1.34
speedup JDK/TOBA	4.54	3.51	5.08	2.98	2.86
speedup JDK/CACAO	11.24	3.90	2.74	7.01	2.30

Die Zeiten im oberen Teil der Tabelle wurden von den Toba-Entwicklern auf einer SparcStation 20 gemessen. Die Zeiten im mittleren Teil habe ich selbst auf einer Alpha-Maschine festgestellt. Dabei habe ich die Laufzeit von CACAO in die drei Komponenten unterteilt:

`load` ist die Zeit, die CACAO braucht, um alle Klassenfiles in den Speicher zu laden.

`compile` ist die Zeit, die fürs Übersetzen des Bytecodes in Maschinensprache notwendig ist.

`run` ist dann die Zeit, die das Programm selbst verbraucht.

In der nächsten Zeile ist die Anzahl der JavaVM-Instruktionen aufgelistet, die vom CACAO bei jedem Programm übersetzt worden sind (die Übersetzungsgeschwindigkeit liegt also irgendwo bei 30000 Instruktionen pro Sekunde).

Im unteren Teil der Tabelle stehen die Beschleunigungsfaktoren der Compiler im Verhältnis zum Java-Bytecode-Interpreter. Bei Programmen, die eine kurze Übersetzungszeit im Verhältnis zum tatsächlichen Programmlauf haben (`JavaLex, Toba`), ist die Beschleunigung wesentlich höher, als bei Programmen, die aufwendiger zu übersetzen sind (`javac, espresso, java_cup`).

9.2 Kleine Testprogramme

Dieses Testprogramm berechnet Primzahlen mit dem Sieb des Eratosthenes. Dabei verwendet es Arrays von verschiedener Größe, um den Effekt des Speicher-Caches auf die Laufzeit zu demonstrieren. Als zweite Referenz habe ich ein equivalentes C-Programm (übersetzt mit `gcc -O3, Version 2.7.2.1`) getestet. Die C-Version ist dabei schneller als CACAO und dieser ist schneller als der Java-Interpreter. Die Laufzeitunterschiede werden umso größer, je kleiner das Array ist, weil die Speicherzugriffe dann nicht mehr so ins Gewicht fallen.

Um den Test fairer zu gestalten, habe ich für den direkten Vergleich von CACAO und C bei CACAO einige Laufzeit-Überprüfungen, die C ja auch nicht durchführt, abgeschaltet (`option: -cbnf`).

Arraygröße	10M	1M	100K	10K	1K
Laufzeiten auf einer 21064A 300MHz (in Sekunden)					
JDK	100.2	92.9	88.6	83.2	79.8
GCC	11.0	3.9	3.3	2.0	1.8
CACAO total	13.38	6.57	5.71	4.80	4.54
run	13.15	6.37	5.51	4.58	4.32
CACAO -cbnf total	12.34	5.30	4.49	3.46	3.28
run	12.19	5.51	4.32	3.31	3.13
Verhältnis der Laufzeiten					
speedup JDK/CACAO	7.48	14.1	15.5	17.3	17.5
speedup JDK/GCC	9.11	23.8	26.8	41.6	44.3
CACAO -cbnf run/GCC	1.11	1.41	1.31	1.66	1.74

Noch ein paar weitere Test zum Vergleich von Java-Interpreter, C und CACAO:

- Ein Programm, das in einer Schleife 100.000.000 Additionen durchführt.
- Der Benchmark `linpack`, der die Fließkommaleistung des Systems testet.

	Addieren	linpack
Laufzeiten auf einer 21064A 300MHz		
JDK	70.7	1.6
GCC	1.4	-
CACAO total	2.70	0.81
run	2.49	0.58
CACAO -cbnf total	2.65	0.33
run	2.49	0.13
Verhältnis der Laufzeiten		
speedup JDK/CACAO	26.2	1.8
speedup JDK/GCC	50.5	-
CACAO -cbnf run/GCC	1.78	-

Kapitel 10

Verwandte Arbeiten

Die Idee, maschinenunabhängige Zwischendarstellungen für Programme zu verwenden (wie es Java tut), ist schon relativ alt und wurde das erste Mal im Jahre 1960 vorgeschlagen [TBS61]. Diese Zwischensprache UNCOL (UNiversal Computer Oriented Language) sollte als Verbindung zwischen dem Front-End und dem Back-End von Compilern eingesetzt werden. Die Einführung einer solchen genormten Schnittstelle würde es ermöglichen, sehr leicht viele Programmiersprachen (ein Front-End für jede) und Zielmaschinen (ein Back-End für jede) zu unterstützen. Tatsächlich fand dieser Vorschlag damals keine große Unterstützung und wurde nie verwirklicht.

Das viel später entwickelte *Amsterdam Compiler Kit* [TvSKS83] greift diese Idee der unabhängigen Zwischendarstellung wieder auf. Die Programme werden hier in die Maschinensprache für eine (virtuelle) Stackmaschine umgewandelt, und dann entweder auf die Zielmaschine übersetzt oder gleich mit einem Interpreter abgearbeitet (zum Debuggen entweder des Übersetzers oder des Programmes selbst).

Die Übersetzung vom Quellprogramm zum tatsächlichen Maschinenprogramm geht dabei in mehreren sprachabhängigen, unabhängigen und maschinenabhängigen Schritten vor sich:

- Preprocessor (sprachabhängig)
- Front end (sprachabhängig)
- Peephole optimizer (unabhängig)
- Global optimizer (unabhängig)
- Back end (maschinenabhängig)
- Target machine optimizer (maschinenabhängig)
- Universal assembler/linker (maschinenabhängig)

Alle diese Abarbeitungsstufen sind nur für den Batch-Betrieb ausgelegt, das heißt, sie nehmen jeweils ein Programm in ihrer Eingabesprache entgegen und erzeugen daraus ein übersetztes oder optimiertes Programm. Übersetzer, die mit dem *Amsterdam Compiler Kit* erstellt werden, verhalten sich also ziemlich genau wie konventionelle Übersetzer, nur eben mit dem Vorteil, den Compilerbauern viel Arbeit zu ersparen.

Im Gegensatz dazu stehen Systeme, bei denen der letzte Schritt, also die explizite Übersetzung in ein maschinenabhängiges Dateiformat, nicht mehr ausgeführt wird. Statt dessen werden die Programme nur in ihrer Zwischendarstellung gespeichert (und auch verteilt, verkauft, etc.).

Der letzte Schritt der Übersetzung passiert entweder gar nie (wenn das Programm von einem Interpreter abgearbeitet werden soll) oder aber erst zur Laufzeit des Programmes (Übersetzung On-the-Fly bzw. Just-In-Time). Für den Benutzer ist es an sich unwichtig, welche dieser beiden Varianten intern im Interpreter/Compiler verwendet wird, der Unterschied liegt nur in der Programmlaufzeit (deshalb wird jetzt auch die Entwicklung von Just-In-Time - Übersetzern für Java sehr vorangetrieben).

Der Vorteil dieses Verfahrens liegt darin, dass die selbe Binärdarstellung eines Programmes auf unterschiedlichsten Maschinen verwendet und außerdem erheblich kompakter gespeichert werden kann als in einer Maschinencode-Darstellung. In [Fra94] behauptet der Autor sogar, dass der Aufwand, der durch die nachträgliche Übersetzung des Programmes in Maschinencode entsteht, durch die kürzere Ladezeit (von langsamen Sekundärspeichern) mehr als kompensiert werden kann.

Die Übersetzung eines stackbasierten Codes in Maschinencode ist nicht ganz einfach, zumindest nicht, wenn das Programm gut und effizient laufen soll. Das Hauptproblem ist vor allem die Abbildung des Stacks auf CPU-Register und die Elimination unnötiger Stackmanipulationen.

Ein gutes Referenzsystem ist RAFTS [Ert92] [Ert96], das Forth (eine rein stackbasierte Programmiersprache) auf Maschinencode übersetzt, und zwar auch erst während der Laufzeit. Dabei wandelt der Compiler die Stackoperationen soweit als möglich auf Registerzugriffe um und produziert intern einen optimierten Maschinencode für alle so übersetzten Programmteile (= words).

Etwas anders als RAFTS führt CACAO alle Optimierungen nicht maschinenabhängig aus, sondern auf einer maschinenunabhängigen Zwischendarstellung, die aus Befehlen, die Pseudoregister als explizite Operanden haben, besteht. Erst im zweiten Schritt (nach der Optimierung) werden diese Befehle tatsächlich auf Maschinencode umgewandelt. Das hat den Zweck, CACAO etwas einfacher auf andere Maschinen portieren zu können.

Dem CACAO aber noch ähnlicher ist der Forth-to-C - Compiler `f2c` [Ert96]. Dieser wandelt einen stackbasierten Forth-Code in C-Code um und benutzt dann einen optimierenden Compiler für die endgültige Übersetzung in Maschinensprache.

Diese Zwischendarstellung in C entspricht in vieler Hinsicht der internen Darstellung wie sie CACAO benutzt. Die Einträge am Stack werden dort auf lokale C-Variablen (wie auf die Pseudoregistern in CACAO) abgebildet, und die Sprungbefehle einfach auf `goto`-Anweisungen. Ebenso wie bei CACAO ist diese Darstellung systemunabhängig und portabel, allerdings nur mit großem Aufwand weiter zu übersetzen, also nicht so geeignet, einen Just-In-Time - Compiler zu implementieren.

Kapitel 11

Zusammenfassung und Ausblick

Der Just-In-Time - Compiler CACAO übersetzt Programme, die als Klassenfiles in Java-Bytecode vorliegen, während des Programmlaufes in Maschinencode und führt diesen aus. Die Übersetzung einzelner Java-Methoden passiert erst zum letztmöglichen Zeitpunkt (nämlich bei deren ersten Aufruf) um die unnötige Übersetzung unbenutzter Programmteile zu verhindern (von einem Java-Programm werden in der Regel nur ein Bruchteil der vorhanden Bibliotheksfunktionen tatsächlich verwendet). Der Compiler führt dabei einige Optimierungen durch, wie: Zusammenfassen mehrerer JavaVM-Instruktionen, Umwandeln des Stacks auf Register, schneller Aufruf virtueller Methoden.

In Summe gesehen, bewirkt die Verwendung von CACAO eine signifikante Beschleunigung gegenüber der Interpretation der JavaVM-Programme mit dem JDK (Java Development Kit). Diese Verbesserung ist natürlich umso größer, je länger die verwendeten Java-Programme im Verhältnis zu ihrer Codelänge laufen, weil da der Aufwand, der durch die einmalige Übersetzung entsteht, nicht mehr so ins Gewicht fällt.

Im Vergleich zu Programmen, die in C geschrieben sind, laufen Java-Programme natürlich immer noch etwas langsamer. Dieser Geschwindigkeitsunterschied um den Faktor 1.5 (ungefähre Schätzung auf Grundlage meiner Benchmarks) entsteht vor allem durch die Übersetzung zur Laufzeit und das komfortable Speichermanagement der JavaVM (Die automatische Speicherfreigabe kostet auf jeden Fall mehr Zeit als explizite Freigabe, ist aber viel sicherer).

Um das System CACAO in zukünftigen Versionen noch weiter zu beschleunigen, gäbe es prinzipiell zwei Möglichkeiten:

- besseren Maschinencode erzeugen
- schneller zu übersetzen

Diese beiden Varianten stehen leider zueinander in einem grundsätzlichen Gegensatz: Für einen besseren Maschinencode sind Optimierungstechniken notwendig, die selber wiederum recht aufwendig sind und die gesammte Laufzeit möglicherweise sogar erhöhen.

Am besten wäre natürlich ein System, das für jede JavaVM-Methode diejenige Optimierungsstufe wählt, die die Gesamtlaufzeit (= Übersetzen + Ausführen) minimiert. Neben der Möglichkeit, die Entscheidung der Optimierungsstufe dem Benutzer zu überlassen, könnte sie das Systems selbst mit Hilfe von Heuristiken treffen (also irgendwie schätzen). Zum Beispiel könnten dabei Faktoren berücksichtigt werden, wie:

- Größe der Methode
- Zeitpunkt des ersten Aufrufes
- Konstruktormethode oder nicht
- Anzahl der Parameter

Abschließend möchte ich sagen, dass die Entwicklung effizienter Just-In-Time-Compiler für Java diese Programmiersprache durchaus konkurrenzfähig im Laufzeit-Vergleich mit C oder C++ macht. Java hat aber den gewaltigen Vorteil, dass die in Java-Bytecode übersetzen Programme auf allen Architekturen ohne Änderung laufen können, und dass sie viel einfacher zu programmieren und zu testen sind.

Vielleicht (und ich hoffe es) wird Java in Zukunft so zur Standard-Programmiersprache, wie es C heute ist.

Anhang A

Implementierungsdetails

In diesem Kapitel beschreibe ich die Struktur von CACAO und die wichtigsten Designentscheidungen, die zu dieser Programmstruktur geführt haben.

A.1 Die Struktur des Programmes

Bei der Aufteilung des Programmes in einzelne Module war für mich der entscheidende Faktor nicht unbedingt die Einsparung von Übersetzungszeit (moderne Rechner sind sowieso schnell genug, viele tausend Zeilen C-Code im Nullkommanichts zu übersetzen), sondern klare Schnittstellen zwischen den Modulen (von denen es außerdem nicht zu viele geben sollte) zu erreichen.

Aus diesem Grund ist der Modul `compiler.o` der bei weitem größte, aber gerade hier sind die einzelnen Programmteile so voneinander abhängig, dass eine weitere Aufteilung (wobei natürlich wieder Headerfiles nötig werden, etc.) nicht mehr sinnvoll ist.

Das Programm besteht jetzt also aus folgenden Teilen:

`main.c` enthält die Funktion `main`. Hier werden die Command-Line-Parameter ausgewertet und alle nötigen Teilsysteme initialisiert und gestartet.

`tables.c` Dieser Modul verwaltet die Symboltabellen, den Heap, und stellt Funktionen zum Lesen von Klassenfiles zur Verfügung.

`loader.c` Hier stehen alle Funktionen, mit denen Klassenfiles gelesen und in interne Datenstrukturen umgewandelt werden können.

`compiler.c` Das ist das eigentliche Herz des Systems. Der Just-In-Time - Compiler wird immer gestartet, wenn eine Java-Methode aufgerufen wird, und für diese noch kein Maschinencode vorliegt.

`builtin.c` enthält Funktionen für JavaVM-Operationen, die nicht direkt als Maschinencode *inline* eingefügt werden können.

`native.c` enthält alle `native` Methoden.

`asmpart.s` ist die Schnittstelle zwischen Java-Methoden und C-Funktionen.

Bei den Modulen `compiler` und `native` habe ich den C-Sourcecode der besseren Übersichtlichkeit halber in verschiedene Dateien verteilt, die zur Übersetzung mit `#include` in den Hauptmodul integriert werden. Diese Teile sind in den Verzeichnissen `comp/` bzw. `nat/` zu finden.

A.2 Die Toolbox

Weil beim Programmieren in C immer wieder die selben grundlegenden Problemstellungen auftregen (Listen, Speicherverwaltung, Protokollierungsdateien, binäre Bäume, etc.) habe ich alle programmunabhängigen Teile in einer eigenen Toolbox untergebracht. Alle Module dafür befinden sich im Verzeichnis `toolbox`, es gibt auch ein eigenes `Makefile`, das die Datei `toolbox.a` erzeugt. Die Headerfiles für die Toolbox sind alle auf `.h` endenden Dateien, entsprechende Dokumentation ist ebenfalls in diesem Verzeichnis zu finden.

A.3 Der Headerfile-Generator

Die `native` Methoden, die von CACAO zur Verfügung gestellt werden, müssen natürlich ganz genau auf die benutzten Java-Basisklassen abgestimmt werden (und umgekehrt). Weil diese Methoden durch C-Funktionen implementiert werden, brauche ich von C aus einen Zugriff auf die wichtigsten Java-Objekte. Die Struktur der Objekte ist aber in Java selbst definiert, deshalb benötige ich irgendein Werkzeug, das für jede Java-Klasse eine entsprechende C-Struktur erzeugt, und außerdem auch gleich eine Tabelle aller verwendeten `native` Methoden.

Der Headerfile-Generator besteht aus ein paar der oben schon erwähnten Module (`tables.s, loader.c`), plus einem zusätzlichen Modul, nämlich `headers.c`, der die Stelle von `main.c` einnimmt.

Die Vorgangsweise zum Erzeugen eines lauffähigen CACAO ist folgende:

- Übersetzen von `tables.c loader.c global.c` und `headers.c`
- Linken der Module zum Programm `cacaoh` (cacao - headers).
- Starten des `cacaoh`, der die beiden Dateien `nativetypes.hh` und `nativetable.hh` erzeugt.
- Übersetzen des Modules `native.c`, der diese Dateien benötigt.

- Übersetzen der restlichen Module (wäre auch schon vorher möglich)
- Linken aller Module (außer `headers.o`) zum lauffähigen `cacao`.

Ich hatte zwar ursprünglich nicht vor, ein separates Programm für diese Aufgabe zu programmieren. Der `cacao` selbst, mit einer speziellen Option gestartet, sollte die Dateien generieren. Aber dazu müsste er sich praktisch selbst verändern (also seine eigenen Sourcen umschreiben), und das hat sich dann als etwas zu umständlich herausgestellt.

A.4 Einbauen von `native` Methoden

Um eine neue Methode in das System einzubauen, muss diese Methode zuerst im entsprechenden Klassenfile deklariert werden (wenn man eine vorhandene Bibliothek von Java-Klassen verwendet, dann entfällt dieser Schritt).

Beim Start des Headerfile-Generators werden (als Commandline-Argumente) die Namen aller Klassen angegeben, für die Strukturinformationen und die `native`-Headers generiert werden sollen. Diese Information wird in der Datei `nativetypes.hh` als C-Code gespeichert, von wo die C-Prototypes dann auch in die entsprechenden Sourcefiles kopiert werden können (alle Dateien im Verzeichnis `nat/` enthalten ausschließlich diese Implementierungen). Die zweite generierte Datei `nativetable.hh` enthält nur eine vollständige Tabelle aller vorhandenen `native` Methoden. Diese Tabelle benötigt CACAO um die entsprechenden Funktionen mit ihrem Namen zu finden.

Für jede solchermaßen deklarierte Funktion muss jetzt noch eine Implementierung programmiert werden. Die Namenskonventionen kann ich am besten anhand eines Beispieles erklären. Die Methode `hashCode` der Klasse `java.lang.Object` würde in etwa so programmiert werden:

```
/* FILE: nat/lang.c */

/* Code fuer die Methode hashCode von java.lang.Object */

s4 java_lang_Object_hashCode (java_lang_Object *this)
{
    u8 k = (u8) this;
    return (this & 0x7fffffff);

}
```

Dabei ist `java_lang_Object` der Typ für ein Objekt der Klasse, der `this`-Zeiger ist bei allen nicht-`static` Methoden notwendig. Ich habe alle Funktionen für je ein Java-Package in einer eigenen C-Datei untergebracht, und zwar in `nat/<packagename>.c`, diese Dateien werden von `native.c` beim Übersetzen eingebunden. Wenn noch zusätzliche Packages (außer `java.lang.*`, `java.io.* und java.util.*`) eingebunden werden sollen, müssen deren Implementierugen in dieser Datei mit `#include` zusätzlich angegeben werden.

Anmerkung: `native` Methoden dürfen im Fehlerfalle eine Exception werfen. Dazu muss ein Exception-Objekt erzeugt, und der Zeiger darauf der globalen Variablen `exceptionptr` zugewiesen werden. Dann muss die Funktion zum Aufrufer zurückkehren (der Rückgabewert der Funktion selbst wird ignoriert).

A.5 Die Laufzeitbibliothek

Die `native` Methoden sind zwar hauptsächlich für die Implementierung der Zugriffe auf das Betriebssystem konzipiert, aber manchmal ist es notwendig, von diesem Methoden aus auch auf die internen Java-Datenstrukturen zuzugreifen (etwa das Erzeugen von neuen Objekten, oder Aufrufen von Java-Methoden).

Zu diesem Zweck gibt es eine Handvoll C-Funktionen in den Modulen `builtin.c` und `native.c`. Für eine genauere Beschreibung möchte ich auf die Headerfiles und die Kommentare in den Modulen selbst verweisen.

Anhang B

Aufruf von CACAO

CACAO wird vom Betriebssystem aus mit dem Kommando:

`cacao [options] classname [program arguments]`

aufgerufen. In der angegebenen Klasse wird sofort die Methode `main` mit der exakten Signatur "`public static void main(String[] argv)`" gestartet. Dieser Methode werden dann die zusätzlich angegebenen Argumente als String-Array übergeben. Die Optionen für CACAO sind großteils mit denen des Java-Interpreters (`java`) von SUN identisch, aber mit einigen wichtigen Abweichungen und vielen zusätzlichen Möglichkeiten.

Hier ist eine Aufstellung der Optionen, die CACAO zur Zeit erkennt:

`-classpath path`
: Setzt den Suchpfad für die Klassenfiles. Normalerweise gibt die Umgebungsvariable `CLASSPATH` an, wo CACAO suchen soll, aber diese Option hat höhere Priorität.

`-Dpropertyname=value`
: Setzt einen Eintrag in der "system property list" auf einen Wert.

`-ms initmem[k|m]`
: Spezifiziert die Größe des Heap bei Programmstart. Sobald das System die angegebenen Menge Speicher verbraucht hat, wird das erste "Garbage Collection" gestartet.

`-mx maxmem[k|m]`
: Spezifiziert die maximale Größe des Heap. Wenn diese Option nicht angegeben wird, dann wird eine Größe von 16 MB angenommen.

`-noasyncgc` ohne Effekt

`-noverify` ohne Effekt

`-oss stacksize[k|m]` ohne Effekt

`-ss stacksize[k|m]` ohne Effekt
: Diese Optionen haben keinen Auswirkung auf den Ablauf von CACAO und dienen nur der Kompatibilität mit dem `java`-Interpreter.

`-v,-verbose`
: CACAO gibt jedesmal eine Meldung aus, wenn er eine Klasse lädt.

`-verbosegc`
: CACAO gibt jedesmal eine Meldung aus, wenn er eine Garbage Collection durchführt.

`-verbosecall`
: Es werden alle Methodenaufrufe mitprotokolliert.

`-time`
: Gibt nach Beendigung des Programmes eine genaue Abrechnung über die verbrauchte CPU-Zeit aus.

`-log logfile`
: Alle Meldungen von CACAO werden anstatt auf "stdout" auf die angegebenen Datei geschrieben.

`-c[b][n][f]`
: Schaltet verschiedene Laufzeit-Checks ab.

  `n(ull)` Bei Methodenaufrufen oder Zugriffen auf Objekte wird keine `NullPointerException` mehr erkannt.

  `b(ounds)` Bei Arrayzugriffen wird die Überprüfung der Grenzen abgeschaltet.

  `f(loats)` Alle Fließkommaberechnung setzen voraus, dass die Operanden gültige, endliche Werte enthalten.

  Diese Optionen können die Geschwindigkeit des Programmes unter Umständen stark beschleunigen. Der Preis dafür ist, dass das Programm, anstatt eine Exception zu erzeugen, ganz einfach abstürzen wird (sollte nur bei wirklich gut ausgetesten Programmen verwendet werden).

`-l`
: Lädt die angegebenen Klassen nur in den Speicher, ohne aber die Methode `main` zu starten (vor allem im Zusammenhang mit einigen der folgenden Optionen nützlich).

`-m methodname`
: Compiliert nur die gewünschte Methode, ohne sie allerdings zu starten (ebenfalls im Zusammenhang mit den folgenden Optionen zu verwenden).

`-s[m][c][a][s][i][u]`
: Diese Optionen sind hauptsächlich für das Debuggen von CACAO selbst nützlich. Der Compiler kann verschiedene interne Datenstrukturen in abdruckbarer Form ausgeben:

  `m(ethod)` Eine Aufstellung aller Methoden und Felder einer Klasse.

  `c(onstant)` Alle Einträge des "Constant Pool".

  `s(tack)` Den genauen JavaVM-Stack bei jeder übersetzten JavaVM-Instruktion.

  `i(ntermediate)` Den Zwischencode der Methoden.

  `a(ssembler)` Ein Assembler-Listing der übersetzten Methoden.

  `u(nicode)` Die gesammte HASH-Tabelle mit allen eingetragen Unicode-Symbolen.

Anhang C

Zwischencode-Befehle

CACAO verwendet für die interne Darstellung der Programme einen eigenen Zwischencode. In diesem Anhang steht eine Auflistung aller dieser Befehle und deren mögliche Parameter.

`LOADCONST` `(Typ) #Wert -> Ziel`
: Schreibt einen konstanten Wert des gewünschten Typs (`INT, LONG, FLOAT, DOUBLE, ADDRESS`) in ein Pseudoregister.

`MOVE` `(Typ) Quelle -> Ziel`
: Kopiert einen Wert aus einem Pseudoregister in ein anderes. Dabei müssen alle beiden Register vom angegebenen Typ sein.

`OP1` `(Operator) Quelle -> Ziel`
: Dieser Befehl führt eine Operation mit einem Quellpseudoregister und einem Zielpseudoregister durch. Die genaue Spezifikation dieser Operation wird durch den zusätzlichen Parameter angegeben (siehe Tabelle). Dieser Parameter bestimmt auch, welche Typen die Quell- und Zielregister haben müssen.

`OP2` `(Operator) Quelle1, Quelle2 -> Ziel`
: Wie OP1, aber mit 2 Quellregistern.

`OP3` `(Operator) Quelle1, Quelle2, Quelle3 -> Ziel`
: Wie OP1, aber mit 3 Quellregistern.

`MEM` `(GETFIELD) (Typ) Offset (Quelle) -> Ziel`
: Holt einen Wert aus dem Speicher. Die Adresse für den Zugriff ergibt sich aus der Adresse im Quellregister plus dem angegebenen Offset. Der geladene Wert kommt dann ins Zielregister. Dabei muss das Quellregister vom Typ `ADDRESS` und das Zielregister vom angegebenen Typ sein.

`MEM` `(PUTFIELD) (Typ) Quelle2 -> Offset (Quelle1)`
Wie `MEM (GETFIELD)`, aber hier wird der Wert aus dem Quellregister 2 in den Speicher geschrieben, der durch dem Inhalt von Quellregister 1 plus den Offset adressiert wird.

`BRA` `(Operator) Sprungziel Quelle1 Quelle2 -> Ziel`
Für alle Arten von Sprung- bzw. Verzweigungsbefehlen, inklusive Methodenbeendigung. Die Art und Anzahl der Quell- und Zielregister hängt vom verwendeten Operator ab (siehe Tabelle). Das Sprungziel ist (wenn vorhanden) eine Referenz auf einen der Blöcke in der Methode.

`TABLEJUMP` `(Sprungtabelle) Quelle`
Für Sprünge mit einer Sprungtabelle (wird eher selten gebraucht). Dabei wird der Inhalt vom Quellregister als Index in die Sprungtabelle angesehen (bei 0 beginnend) und an der dort angegebenen Stelle hingesprungen. Weil zum Zeitpunkt, an dem der Zwischencode erzeugt wird, die tatsächlichen Sprungziele im Maschinencode noch nicht feststehen, werden in dieser Zwischendarstellung vorerst nur Referenzen auf Blöcke eingetragen.

`METHOD` `(Operator) (Descriptor) Quelle1, ... -> Ziel, Exception`
Mit dieser Instruktion werden alle Arten von Methodenaufrufen getätigt. Sie erlaubt eine unbegrenzte Anzahl von Parametern (müssen vorher in Pseudoregistern bereitgestellt werden) und schreibt den Rückgabewert der Methode in das angegebene Zielregister. Wenn die aufgerufene Methode ein Exception erzeugt hat, dann steht der Zeiger auf das Exception-Objekt nachher im angegebenen Register (falls keine Exception aufgetreten ist, dann wird hier NULL zurückgegeben). Der Descriptor spezifiziert die aufzurufende Methode genauer (Art und Anzahl der Parameter, Name der Methode).

Mögliche Operatoren:

`INVOKESTATIC` Aufruf einer `static` Methode. Das Sprungziel ist dabei schon zur Übersetzungszeit bekannt.

`INVOKESPECIAL` Ruft eine nicht-`static` Methode auf. Aber auch hier wird eine Methode gleich direkt aufgerufen ohne erst zur Laufzeit das Aufrufsziel festzustellen (für Initializer in Java benötigt). Der Wert des Quellregisters 1 wird dabei zum Wert des `this`- Zeigers in der aufgerufenen Methode.

`INVOKEVIRTUAL` Ruft eine nicht-`static` Methode mit Hilfe des normalen Aufruf-Verfahrens mit Virtual Function Tables auf.

INVOKEINTERFACE Ruft eine Interface-Methode auf. Das Quellregister 1 enthält den zu verwendenden `this`-Zeiger, über den der Aufruf stattfinden soll.

DROP `Pseudoregister`
Das ist eigentlich gar keine Instruktion, sondern nur ein Hinweis an den nachfolgenden Registerallokator, dass das angegebene Pseudoregister ab hier nicht mehr gebraucht wird, und das ihm zugewiesene CPU-Register wieder anderwertig verwendet werden kann.

ACTIVATE `Pseudoregister`
Ist das Gegenstück zu DROP und sagt dem Registerallokator, dass ein Pseudoregister ab hier aktiv ist.

C.1 Tabellen der Operatoren

Hier ist eine Auflistung der Operatoren, die im Zwischencode auftreten können. Die Semantik der Operationen entspricht genau der Beschreibung, die in der JavaVM-Spezifikation [LY96] gegeben wird. Die Stackeffekte sind dabei aber durch explizite Zugriffe auf Pseudoregister ersetzt.

OP1-Operatoren	Typ Quelle	Typ Ziel
INEG	INT	INT
I2L	INT	LONG
I2F	INT	FLOAT
I2D	INT	DOUBLE
INT2BYTE	INT	INT
INT2CHAR	INT	INT
INT2SHORT	INT	INT
LNEG	LONG	LONG
L2I	LONG	INT
L2F	LONG	FLOAT
L2D	LONG	DOUBLE
FNEG	FLOAT	FLOAT
F2I	FLOAT	INT
F2L	FLOAT	LONG
F2D	FLOAT	DOUBLE
DNEG	DOUBLE	DOUBLE
D2I	DOUBLE	INT
D2L	DOUBLE	LONG
D2F	DOUBLE	FLOAT
ARRAYLENGTH	ADDRESS	INT

OP2-Operatoren	Typ Quelle 1	Typ Quelle 2	Typ Ziel
IADD	INT	INT	INT
ISUB	INT	INT	INT
IMUL	INT	INT	INT
ISHL	INT	INT	INT
ISHR	INT	INT	INT
IUSHR	INT	INT	INT
IAND	INT	INT	INT
IOR	INT	INT	INT
IXOR	INT	INT	INT
LADD	LONG	LONG	LONG
LSUB	LONG	LONG	LONG
LMUL	LONG	LONG	LONG
LSHL	LONG	INT	LONG
LSHR	LONG	INT	LONG
LUSHR	LONG	INT	LONG
LAND	LONG	LONG	LONG
LOR	LONG	LONG	LONG
LXOR	LONG	LONG	LONG
LCMP	LONG	LONG	INT
FADD	FLOAT	FLOAT	FLOAT
FSUB	FLOAT	FLOAT	FLOAT
FMUL	FLOAT	FLOAT	FLOAT
FDIV	FLOAT	FLOAT	FLOAT
FREM	FLOAT	FLOAT	FLOAT
FCMPL	FLOAT	FLOAT	INT
FCMPG	FLOAT	FLOAT	INT
DADD	DOUBLE	DOUBLE	DOUBLE
DSUB	DOUBLE	DOUBLE	DOUBLE
DMUL	DOUBLE	DOUBLE	DOUBLE
DDIV	DOUBLE	DOUBLE	DOUBLE
DREM	DOUBLE	DOUBLE	DOUBLE
DCMPL	DOUBLE	DOUBLE	INT
DCMPG	DOUBLE	DOUBLE	INT
IALOAD	ADDRESS	INT	INT
CALOAD	ADDRESS	INT	INT
SALOAD	ADDRESS	INT	INT
BALOAD	ADDRESS	INT	INT
LALOAD	ADDRESS	INT	LONG
FALOAD	ADDRESS	INT	FLOAT
DALOAD	ADDRESS	INT	DOUBLE
AALOAD	ADDRESS	INT	ADDRESS

OP3-Operatoren	Typ Quelle 1	Typ Quelle 2	Typ Quelle 3
IASTORE	ADDRESS	INT	INT
CASTORE	ADDRESS	INT	INT
SASTORE	ADDRESS	INT	INT
BASTORE	ADDRESS	INT	INT
LASTORE	ADDRESS	INT	LONG
FASTORE	ADDRESS	INT	FLOAT
DASTORE	ADDRESS	INT	DOUBLE
AASTORE	ADDRESS	INT	ADDRESS

BRA-Operatoren	Quelle 1	Quelle 2	Typ Ziel	Sprungziel?
GOTO	-	-	-	ja
JSR	-	-	ADDRESS	ja
RET	ADDRESS	-	-	nein
IFEQ	INT	-	-	ja
IFLT	INT	-	-	ja
IFLE	INT	-	-	ja
IFNE	INT	-	-	ja
IFGT	INT	-	-	ja
IFGE	INT	-	-	ja
IFEQL	LONG	-	-	ja
IFNULL	ADDRESS	-	-	ja
IFNONNULL	ADDRESS	-	-	ja
IF_ICMPEQ	INT	INT	-	ja
IF_ICMPNE	INT	INT	-	ja
IF_ICMPLT	INT	INT	-	ja
IF_ICMPGT	INT	INT	-	ja
IF_ICMPLE	INT	INT	-	ja
IF_ICMPGE	INT	INT	-	ja
IF_ACMPEQ	ADDRESS	ADDRESS	-	ja
IF_ACMPNE	ADDRESS	ADDRESS	-	ja
IRETURN	INT	-	-	nein
LRETURN	LONG	-	-	nein
ARETURN	ADDRESS	-	-	nein
FRETURN	FLOAT	-	-	nein
DRETURN	DOUBLE	-	-	nein
RETURN	-	-	-	nein

Anmerkung: Die `IFEQL`-Instruktion ist in der JavaVM-Spezifikation nicht definiert. CACAO benötigt sie aber für seinen Zwischencode um die Divisor-Überprüfung bei `LONG`-Divisionen durchzuführen. Die Semantik entspricht dabei bis auf den Typ exakt der `IFEQ`-Instruktion.

Literaturverzeichnis

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bha95] Dileep P. Bhandarkar. *Alpha Implementation and Architekture, Complete Reference and Guide.* Digital Press, 1995.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98 – 105, 1982.

[Cor92] Digital Equipment Corporation. *Alpha Architecture Handbook.* Digital Press, 1992.

[CWF92] Todd A. Proebsting Christopher W. Fraser, David R. Hanson. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Language and Systems*, pages 213 – 226, 1992.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, 1992.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technische Universität Wien, April 1996.

[Fla96] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996.

[Fra94] Michael Franz. *Code Generation On the Fly: A Key for Portable Software.* PhD thesis, ETH Zürich, 1994.

[GR83] Adele Goldberg and David Robson. *SMALLTALK-80, The Language and its Implementation.* Addison-Wesley, 1983.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach. second edition.* Morgan Kaufman Publishers, 1996.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection.* John Wiley & Sons, 1996.

[KD95] Jan Vitek Karel Driesen, Urs Hölzle. Message dispatch on pipelined processors. In *ECOOP '95 – Object-Oriented Programming*, pages 251 – 282, 1995.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, pages 419 – 429, 1983.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[Muc86] Phillip B. Gibbons & Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, pages 11 – 16, 1986.

[PBT89] Ken Kennedy Preston Briggs, Keith D. Cooper and Linda Torczon. Coloring heuristics for register allocation. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275 – 284, 1989.

[RS95] R.T. Witek R.L. Sites. *Alpha AXP Architecture Reference Manual.* Digital Press, 1995.

[TBS61] Jr. T. B. Steel. A first version of uncol. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, pages 371 – 377, 1961.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 16(9):654–660, September 1983.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engeneering Symposium on Practical Software Development Environments*, pages 157 – 167, 1984.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87 Proceedings*, pages 227 – 242, 1987.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *6th International Conference CC '96*, pages 307 – 325, 1996.