Optimierung mittels der Auswahl von String Repräsentationen in Java Bytecode

Markus Wondrak
Goethe Universität Frankfurt am Main
http://www.sepl.informatik.uni-frankfurt.de

August 5, 2014

Contents

List of Figures

List of Tables

Abstract

Es geht um blibla blubb

Chapter 1

Einleitung

In dieser Arbeit soll untersucht werden, ob es möglich ist...

Chapter 2

Werkzeuge

In den folgenden Abschnitten sollen die verwendeten Werkzeuge kurz vorgestellt werden. Dabei handelt es sich zum einen um den Java Bytecode, als auch um die Software Bibliothek *WALA*, auf deren API das von mir entwickelte System basiert.

2.1 Java Bytecode

Die Plattformunabhängigkeit, die in Java geschriebenen Programmen zugesprochen wird, ist vorallem mit der Rolle der Java Virtual Machine zu erklären. Java Programme werden in einen Zwischencode, den Java Bytecode, übersetzt, welcher von der System spezifischen JVM ausgeführt wird. Dabei ist Programmiersprache Java nicht die einzige in Bytecode übersetzbare Sprache. Es existieren neben den bekanntesten Scala, Jython, Groovy, JavaScript noch viele weitere. Einmal in Bytecode übersetzt in diesen Sprachen geschriebene Programme auf jeder der Java Spezifikation entsprechenden JVM aufführen.

Bytecode ist eine Sammlung von Instruktionen welche durch *opcodes* von 2 Byte Länge definiert werden. Zusätzlich können noch 1 bis n Parameter verwendet werden. Die Sprache ist Stack-orientiert, das bedeutet, dass von Operationen verwendete Parameter über einen internen Stack übergeben werden. Als Beispiel dient der folgende Bytecode:

```
ICONST 5     // legt den konstanten int Wert 5 auf den
   Stack
ILOAD 1      // läd die lokale integer Variable 1 und legt
    sie auf den Stack
```

```
IADD         // addiert die ersten beiden Werte auf dem
   Stack und legt das Ergebnis auf den Stack
ISTORE 2     // speichert den Wert auf dem Stack in der
   Variable 2
```

Dabei existiert der Stack nur als Abstraktion für den eigentlichen Prozessor im Zielsystem. Wie die jeweilige JVM den Stack in der Ziel Plattform umsetzt ist nicht definiert. Die Instruktionen lassen sich in folgende Kategorien einordnen:

- Laden und Speichern von lokalen Variablen (`ILOAD`, `ISTORE`)
- Arithmetische und logische ausdrücke (`IADD`)
- Object Erzeugung bzw. Manipulation (`NEW`, `PUTFIELD`)
- Stack Verwaltung (`POP`, `PUSH`)
- Kontrollstruktur (`IFEQ`, `GOTO`)
- Methoden Aufrufe (`INVOKEVIRTUAL`, `INVOKESTATIC`)

2.2 WALA

Bei *WALA* handelt es sich um die "T.J. Watson Library for Analysis". Eine von ehemals von IBM entwickelte Bibliothek für die statische Codeanalyse von Java- und JavaScript Programmen. Das Framework übernimmt dabei das Einlesen von *class* Dateien und stellt eine Repräsentation, die sogenannte *Intermediate Representation*, des Bytecodes zur Verfügung. Diese IR stellt die zentrale Datenstruktur dar und soll in diesem Abschnitt detailiert beschrieben werden.

Für die Manipulation des Bytecodes existiert innerhalb des Frameworks ein Unterprojekt, das diese Aufgabe übernimmt: Shrike. Im Zweiten Abschnitt soll diese API kurz vorgestellt werden.

2.2.1 IR

Die *Intermediate Representation* (IR) eine Abstraktion zum Stack basierten Bytecode. Ein IR ist in Single Static Assignment Form, welche sich dadurch auszeichnet, dass jeder Variablen immer genau **einmal** ein Wert zugewiesen

wird. Zusätzlich besteht die IR aus dem Kontroll Fluß Graphen der Methode, welcher wiederrum aus Basic Blocks zusammengesetzt ist. Ein Basic Block ist eine Zusammenfassung von aufeinander Folgende Instruktionen, welche in jedem Fall nach einander ausgeführt werden.

Die Variablen innerhalb des IRs nennt WALA *value numbers*. Diese beziehen sich immer auf eine Referenz, allerdings kann sich eine Referenz sich auf mehrere tatsächliche value numbers in der IR beziehen. Dies folgt aus der SSA Form, wird eine Variable im Bytecode zweimal ein Wert zugewiesen, wird diese doppelte Zuweisung in der SSA Form durch das Einführen einer neuen value number entfernt. Die Operationen werden auch nur mit Bezug auf die value numbers beschrieben.

Da die Zwischendarstellung vom Stack abstrahieren soll, werden auch alle Operationen, die den Stack betreffen (wie z.B. `LOAD`, `STORE`, `PUSH` oder `POP`) nicht nicht mit in diese Repräsentation übernommen. Dabei werden die Bytecode Indices der überig gebliebenen Instruktionen berücksichtigt und alle anderen Stellen in dem beinhaltenden Array mit `null` Werten aufgefüllt. Instruktionen werden von Objekten vom Typ `SSAInstruction` und dessen Untertypen dargestellt.

Die Verwaltung der value numbers wird von einem Typ namens `SymbolTable` übernommen. Es kommt bei der IR Erstellung zum Einsatz, wenn bei der Simulation des Bytecodes neue Variablen verwendet werden, um neue value numbers zu erzeugen.

Aufgrund der SSA Form der IR lässt sich für jede value number genau eine Definition bestimmen. Zu diesem Zweck bietet WALA den Typ `DefUse` an, welcher für jedes IR-Objekt erstellt werden kann. Er ermöglicht einen einfachen Zugriff aus die Instruktionen, die eine value number definiert (*def*; z.B. als Rückgabe aus einem Methodenaufruf), und eine Menge an Instruktionen, die die entsprechende value number verwenden (*use*; z.B. als Rückgabewert in einem `RETURN` Statement).

Das IR und das dazugehörige DefUse Objekt werden in dem System internen Datentyp `AnalyzedMethod` zusammengefasst.

Anpassungen

In WALA werden beim Erstellen des IR für alle Konstanten mit demselben Wert dieselbe value numbers erzeugt. Da für die *Analyse* verschiedenen Referenzen getrennt getrennt untersucht werden mussten, wurde für die Erzeugung einer value number für eine Konstante der eingebaute caching

Mechanismus umgangen.

Darüber hinaus war für die *Transformation* die Information nötig, an welcher Stelle im Bytecode eine entsprechende Konstante erzeugt wird (z.B. mittels `LCD`). Um dies zu Erreichen wurde dieser Bytecode Index während dem Durchlaufen der Instrukionen innerhalb der `SymbolTable` gespeichert, sodass er beim Klienten des IRs zur Verfügung steht.

Da diese Änderungen nicht in den Haupt Branch von WALA eingepflegt werden durften, benötigt das System den Fork des WALA Projektes [1].

2.2.2 Shrike

Shrike ist ein Unterprojekt innerhalb des WALA Frameworks. Shrike übernimmt dabei das Lesen und das Schreiben von Bytecode aus bzw. in class Dateien. Dabei wird es zum einen beim Erstellen eines IR aus einer Methode verwendet, zum Anderen bietet es eine "Patch-based" API an um den Bytecode einer eingelesenen Methode zu verändern. Dies geschieht über das Einfügen von `Patch`es, welche über einen entsprechenden `MethodEditor` überall im Bytecode einer Methode eingefügt werden oder auch ursprüngliche Instruktionen komplett ersetzen. Zusätzlich enthält es einen `Verifier`, der erzeugten Bytecode überprüft, so dass ungültige Stack Zustände oder Typfehler noch während der Manipulation erkannt werden können.

In dem von mir entwickelten System werden alle Bytecode Manipulationen mit Hilfe von Shrike umgesetzt.

[1] Dieser ist unter `http://github.com/wondee/WALA` zu finden.

Chapter 3

Analyse

Das Folgende Kapitel beschreibt den Analyse Algorithmus, des von mir entworfenen Systems. Im ersten Abschnitt sollen die verwendeten Datenstrukturen vorgestellt und beschrieben werden. Der zweite Abschnitt beschreibt den eigentlichen Algorithmus.

3.1 Datenstrukturen

Für den Algorithmus wurden zwei Grundlegende Datenstrukturen entworfen. Der *Datenflußgraph* repräsentiert den Datenfluß der Referenzen innerhalb einer Methode und wird im ersten Abschnitt vorgestellt. Zu optimierende Referenzen werden in diesem Graphen mit sogenannten *Labels* versehen. Dieser Datentyp woll im zweiten Abschnitt beschrieben werden.

3.1.1 Datenfluß Graph

Eine auf einem IR basierende Methode wird vom System mittels eines Datenfluß graphen repräsentiert. Dieser wird vor der eigentlichen Analyse aus einer gegeben Methode und einer Menge an initialen Referenzen vom `DataFlowGraphBuilder` erzeugt. Der Graph ist gerichtet und setzt sich aus zwei verschiedenen Knoten zusammen:

- `Reference`: eine value number aus dem IR
- `InstructionNode`: eine Instruktion aus dem IR

Sei im Folgenden der Datenflußgraph $G = (V, E)$, R die Menge aller `Reference` Knoten und I die Menge aller `InstructionNodes`.
Im Graph gilt $\forall(x, y) \in V, (x \in R \wedge y \in I) \vee (x \in I \wedge y \in X)$. Eine Kante $i \in I, r \in R, (i, r)$ beschreibt eine *Definition*, die aussagt, dass die Referenz r durch die Instruktion i definiert wird. Ein Kante $i \in I, r \in R, (r, i)$ ist eine *Benutzung* (im folgenden *Use* genannt).

`Reference` Knoten werden für jede betroffene value number erzeugt. Für die Erstellung von `InstructionNode` Objekten steht die `InstructionNodeFactory` zur Verfügung, die für eine gegebene `SSAInstruction` eine entsprechende `InstructionNode` erstellt. Um für diesselbe `SSAInstruction` immer dasselbe `InstructionNode` Objekt zu garantieren verwendet die Factory einen internen Cache, der eine Abbildung $SSAInstruction \rightarrow InstructionNode$ verwaltet und für jede `SSAInstruction` prüft ob für diese bereits eine `InstructionNode` erstellt wurde.

Die Erstellung eines `DataFlowGraph`s beginnt immer mit einer Menge an initialen `Reference` Objekten. Ausgehend von dieser Startmenge werden über das `DefUse`-Object des betroffenen IRs die Definition und alle Uses in den Datenflußgraphen eingefügt. Algorithmus 1 beschreibt die Erstellung des Graphen.

Algorithm 1 Erstellung des Datenflußgraphen

```
1: q ← new Queue(initialReferences)
2: g ← new DataFlowGraph()
3: while not q.isEmpty() do
4:   r ← q.remove()
5:   if not g.contains() then
6:     def ← defUse.getDef(r)
7:     uses ← defUse.getUses(r)
8:     newInd.add(def)
9:     newInd.add(uses)
10:    r.setDef(factory.create(def))
11:    for ins ∈ defUse.getUses(r) do
12:      r.addUse(factory.create(ins))
13:    end for
14:    for ins ∈ newIns do
15:      q.add(ins.getConnectedRefs())
16:    end for
17:    g.add(r))
18:  end if
19: end while
20: return g
```

Jede `InstructionNode` besitzt eine Definition, die Nummer der Referenz die diese Instruktion erzeugt und eine Liste von Uses, die Nummern der Referenzen die es benutzt. Darüber hinaus noch Informationen zu Bytecode Spezifika, die im Kapitel *Transformation* betrachtet werden.

Für verschiedene `SSAInstruction` typen existieren entsprechende `InstructionNode` Subtypen. Allerdings gibt es auch Typen die nicht einer `SSAInstruction` zugeordnet werden können. Im Folgenden sollen die wichtigsten Knotentypen vorgestellt werden. Es existieren darüber hinaus weitere für die das System zur Zeit keine Unterstützung bietet, da es ausschliesslich für String Typen und komplexe Objekte ausgelegt ist.

MethodCallNode

Eine `MethodCallNode` repräsentiert einen Methoden Aufruf. Es besitzt, wenn vorhanden, eine Definition, welche den Rückgabewert repräsentiert, einen

Receiver, wenn es keine statische Methode ist und eine Liste an Parametern. Zusätzlich die aufgerufene Methode.

ContantNode

Dieser Knoten Typ stellt eine Konstanten Definition dar. Er besitzt ausschließlich die Defintion, welcher Referenz diese Konstante zugewiesen wird. Für diesen Typ existiert keine entsprechende `SSAInstruction`.

ParameterNode

Die `ParameterNode` stellt eine Definition eines Parameters der Methode dar. Wird eine Variable innerhalb der Methode als Parameter in der Methoden Signatur deklariert, wird deren Definition als `ParameterNode` im Datenflußgraphen repräsentiert. Für diesen Typ existiert keine entsprechende `SSAInstruction`.

NewNode

Dieser Typ entspricht einer `NEW` Anweisung, die ein neues Objekt eines gegebenen Typen erstellt. Es besitzt eine Definition und den Typ des instanziierten Objekts.

ReturnNode

`ReturnNode` Typen sind `RETURN` Anweisungen. Die besitzen ausschließlich eine Referenz als Use. Diejenige Referenz, die sie aus der Methode zurückgeben. Dieser Typ kann keine Definition darstellen.

PhiNode

Die `PhiNode` steht für eine ϕ-Instruktion aus dem IR. Sie besitzt eine Referenz als Definition und 2 bis n Uses.

3.1.2 Label

Das *Label* entspricht einer Markierung, mit der Knoten in einem Datenflußgraphen versehen werden können. Dabei steht ein Label (oder `TypeLabel`,

wie der Datentyp im System heißt) für einen Optimierten Typ. Die Semantik hinter einem markierten Knoten ist, dass diese Referenz bzw. Instruktion durch den entsprechenden Optimierten Typ ersetzt werden kann.

Es kann nicht für alle `InstructionNode`s ein Label gesetzt werden. Genauer gesagt lassen sich ausschließlich für die Knotentypen `MethodCallNode`, `NewNode` und `PhiNode` ein Label setzen, da sich nur diese Instruktionen in einen optimierten Typ umwandeln lassen.

Das `TypeLabel` beinhaltet alle Regeln, die für die Verwendung eines Optimierten Typen existieren. Dazu gehören

- der Originale, sowie der Optimierte Typ
- die Methoden für die Optimierungen im optimierten Typ angeboten werden
- alle Methoden die darüber hinaus vom Optimierten Typ unterstützt werden
- Methoden, die den optimierten Typ als Rückgabewert zurückgeben
- kompatible Label

Dabei ist diese Liste bereits eine Abstraktion zu den Methoden, die das Interface besitzt. Im System lassen sich Label Definition auf 2 Arten erstellen:

1. Durch das Implementieren des Interfaces `TypeLabel`
2. Durch das Erstellen einer `.type` Datei

Zwar unterstützt das Kommandozeilen Tool zur Zeit nur die zweite Variante, programmatisch lässt sich allerdings auch die erste Alternative umsetzen. Im Folgenden sollen die beiden Möglichkeiten zur Definition eines `TypeLabel`s betrachtet werden.

Das TypeLabel Interface

Das Interface beinhaltet alle Methoden, die der Analyse- und Transformationsprozess benötigt. In diesem Kapitel sollen zunächst nur die Methode betrachtet werden, die für den Analyse Algorithmus verwendet werden.

`canBeUsedAsReceiverFor(MethodReference)` Legt fest, ob eine markierte Referenz als Receiver für den übergebenen Methodenaufruf dienen kann.

`canBeUsedAsParamFor(MethodReference,int)` Bestimmt, ob eine markierte Referenz als Parameter in dem gegebenen Methodenaufruf an der entsprechenden Stelle (der `int` Parameter) verwendet werden kann.

`canBeDefinedAsResultOf(MethodReference)` Sagt aus, ob die gegebene Methode einen optimierten Typ zurückgeben kann. Dies impliziert, dass der Methodenaufruf selber auch markiert ist.

`findTypeUses(AnalyzedMethod)` Gibt eine Menge an `Reference` Objekten zurück, auf denen innerhalb der gegebenen Methode eine der von der Optimierung betroffenen Methode aufgerufen wird. Für diesen Algorithmus existiert bereits eine Implementierung in der Klasse `BaseTypeLabel`.

`compatibleWith(TypeLabel)` Gibt an, ob das übergebene Label kompatibel mit diesem Label ist.

Alle diese Methoden werden von den `InstructionNode` Implementierungen verwendet. Wie genau das passiert wird im Abschnitt *Algorithmus* beschrieben.

Das .type Dateiformat

Da das implementieren des Interfaces eher komplex ist, wurde für die einfachere Definition eines Types ein Datei Format entwickelt, welches von der Komplexität des Interfaces abstrahieren soll. In dieser werden nicht die Regeln selbst, sondern die Fakten beschrieben, aus denen die Regeln hergeleitet werden können, beschrieben.

Aus einer Datei im `type` Format wird mittels eines internen Parsers ein `TypeLabelConfig` Objekt erzeugt.

3.2 Algorithmus

3.2.1 Ziel

Chapter 4

Transformation

4.1 Lokale Variablen

Chapter 5

Auswertung

Chapter 6

Fazit