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 15, 2014

Contents

List of Figures

List of Algorithms

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 sowohl 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 vor allem 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 können, in diesen Sprachen geschriebene, Programme auf jeder, der Java Spezifikation entsprechenden, JVM ausgeführt werden.

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
```

Figure 2.1: Java Bytecode Beispiel

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 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 detailliert 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) ist 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 Kontrollflussgraphen der Methode, welcher wiederum 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 übrig 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).

Besitzt ein Block im Kontrollflussgraphen mehrere eingehende Kanten und werden aus diesen Vorgänger Blöcken Variablen mitgebracht die synonym in diesem Block verwendet werden, werden in SSA-Form sogenannte ϕ Funktionen verwendet. Eine Instruktion der Form $v_3 = \phi(v_1, v_2)$ sagt aus,

dass im Folgenden die Referenz v_3 sowohl v_1, als auch v_2 sein kann. Da die statische Code Analyse nicht feststellen kann von welchem Block aus dieser Block betreten wurde, werden diese ϕ-Funktionen verwendet, um die Zusammenführungen von mehreren Variablen aus Vorgängerblöcken darzustellen.

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 Instruktionen 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 soll im zweiten Abschnitt beschrieben werden.

3.1.1 Datenfluss Graph

Eine auf einem IR basierende Methode wird vom System mittels eines Datenflussgraphen 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 dieselbe `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`-Objekt 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 4.1 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 Definition, 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, die Übrigen werden im Abschnitt 4.2.1 betrachtet.

`canBeUsedAsReceiverFor(MethodReference)` Legt fest, ob eine markierte Referenz als Empfänger 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 für den Algorithmus hergeleitet werden können, beschrieben.

Aus einer Datei im `type` Format wird mittels eines internen Parsers ein `TypeLabelConfig` Objekt erzeugt, welches als `TypeLabel` Objekt für den Algorithmus fungiert.

Für die inhaltliche Struktur der Datei wurde JSON (JavaScript Object Notation) gewählt eine Darstellung anzubieten, die sowohl für Menschen als auch für das Programm leicht zu lesen und zu verstehen ist. Die Attribute innerhalb der Datei werden im Folgenden beschrieben:

name Der Name des Labels

originalType Der voll qualifizierte Name des zu ersetzenden Typs

optimizedType Der voll qualifizierte Name des zu optimierten Typs

methodDefs Liste von Methoden, diesen wird eine ID vergeben um sie im folgenden über diese ID zu referenzieren. Ein Eintrag in dieser Liste setzt sich zusammen aus:

- **id** eine eindeutige ID für die diese Methode
- **desc** die Beschreibung dieser Methode. Dies ist ein eigenes Objekt und besteht aus den Attributen:
  - **name** der Name der Methode
  - **signature** der Signatur der Methode. Zusammengesetzt aus der Parameterliste und der Rückgabewert. Die Typen müssen dabei in der internen JVM Form angegeben werden. (Beispiel: `"(I)Ljava/lang/String;"` , ein Parameter vom Typ `int` und Rückgabewert vom Typ `java.lang.String`)

effectedMethods Liste von Methoden IDs. Für diese Methoden existieren optimierte Varianten in dem optimierten Typen.

supportedMethods Liste von Methoden IDs. Diese Methoden werden auch vom optimierten Typ unterstützt. Es handelt sich bei diesen aber nicht um Optimierungen.

producingMethods Liste von Methoden IDs. Alle diesen Methoden erzeugen in ihrer optimierten Variante optimierte Typen.

compatibleLabel Liste von Strings. Alle Labels die mit diesem Label kompatibel sind.

parameterUsage Ein Objekt. Dabei ist jeder key die ID einer Methode und der entsprechende value eine Liste von Ganzzahlen. Ein Eintrag bedeutet, dass diese Methode mit einem Optimierten Typ als Parameter mit diesem Index umgehen kann.

staticFactory Ein String. Der Name einer statischen Factory Methode mit einem Übergabeparameter vom Typ des Originalen Typs. Diese muss einen entsprechenden Optimierten Typ zurückgeben.

toOriginalType Ein String, Der Name einer Methode ohne Parameter, die aus dem optimierten Objekt, ein entsprechendes vom Originalen Typ zurückgibt.

3.2 Algorithmus

In diesem Abschnitt soll die Idee hinter dem Analyse Algorithmus sowie dessen Implementierung vorgestellt werden. Hierzu wird zunächst das Konzept der "Bubble" erläutert um danach die Umsetzung dieses Konzepts im eigentlich Algorithmus zu betrachten.

3.2.1 Motivation

Optimierte Referenzen sind im Falle von String Objekten nicht kompatibel mit den Originalen. So treten Probleme in den Folgenden Szenarien auf:

Referenz als Methoden Parameter Die Signatur der optimierten Methode, wird nicht verändert, da es dazu führen würde, dass Klienten Code, der diese Methode weiterhin mit dem originalen Typ aufruft, nicht mehr kompilieren würde.

Referenz als Rückgabewert Ein ähnliches Problem existiert, wenn die Referenz zurückgegeben wird. Die Signatur der Methode definiert den Originalen Typ als Rückgabetyp und das zurückgeben eines anderen Typs als eben dieser definierte würde zu Laufzeitfehlern führen.

Methodenaufruf auf Referenz Wird diese optimierte Referenz als Empfänger von einer Methode verwendet, die nicht zu den optimierten oder unterstützen Methoden dieses optimierten Typs gehört, würde es zu Laufzeitfehlern kommen, da diese aufzurufende Methode nicht im optimierten Typ vorhanden ist.

Feld Zugriff (`GETFIELD`, `PUTFIELD`) Wird die optimierte Referenz in ein oder aus einem Feld innerhalb eines Objektes (oder in ein statisches Feld einer Klasse) gesetzt, stimmt auch in diesem Szenario der Typ des optimierten und des originalen Objekts nicht überein.

Array Zugriff Bei dem Zugriff auf ein Array, sowohl lesend als auch schreibend, existiert eine Unstimmigkeit mit dem Typ des Arrays.

Referenz Aufruf Parameter Wird die Referenz als Parameter für einen Methoden Aufruf verwendet und diese Methode ist nicht in der Label Definition als Methode deklariert, die mit einem optimierten Typ umgehen kann, dann erwartet diese Methode den originalen Typ und nicht den optimierten.

Erweitert allerdings der optimierte Typ seinen originalen Typ so wäre, durch den Polymorphismus, der optimierte Typ genauso wie der originale verwendbar. Allerdings ist der Typ `java.lang.String` final, was bedeutet, dass von diesem Typ nicht abgeleitet werden kann. Darüber hinaus bieten sich Ableitungen für Optimierungen nicht an, da das dynamische dispatchen zusätzlichen Aufwand für die JVM darstellt, da zunächst die Implementierung des Methode zu lokalisieren.

Aus diesem Grund müssen in den Code Konvertierungen in und vom optimierten Typ eingefügt werden. Eine Konvertierung zum optimierten Typ muss vor dem zu optimierenden Methodenaufruf erfolgen. Eine Umwandlung vom optimierten Typ allerdings muss vor einer nicht kompatiblen Benutzung dieser Referenz erfolgen um bei dieser den originalen Typ zu verwenden.

3.2.2 Die "Bubble"

Da Konvertierungen zusätzliche Laufzeit erfordern, muss es das Ziel sein die Anzahl der durchgeführten Konvertierungen zu minimieren. Dies wird erreicht indem man den Bereich, in dem ein optimierter Typ statt des originalen innerhalb der Methode verwendet maximiert. Dieser Bereich, in dem ein optimierter, statt des originalen, Typs für eine Referenz verwendet wird, wird im Folgenden *Bubble* genannt.

Die Bubble entsteht mittels der Markierung von Knoten im Datenflussgraphen. Es wird für jede gegebene Instanz vom Typ `TypeLabel` eine Bubble auf dem Graphen erzeugt. Dabei kann ein Knoten immer nur mit einem Label markiert sein, daher kann ein Knoten immer nur Teil einer Bubble sein.

Ziel des Algorithmus ist es diese Bubble so groß wie zu definieren. Als Anfangszustand werden alle zu optimierenden Methodenaufrufe mit dem zu verarbeitenden Label markiert.

3.2.3 Umsetzung des Algorithmus

Als Eingabe für den Algorithmus dient ein Objekt vom Typ `DataFlowGraph`, der den bereits beschriebenen Datenflussgraphen darstellt, mit initial markierten `References`. Die Implementierung ist in der Klasse `LabelAnalyzer` zu finden.

Algorithmus 2 beschreibt die Implementierung der Vererbung des Labels.

Algorithm 2 Vererbung des Labels

```
 1: q ← new Queue(initialReferences)
 2: g ← new DataFlowGraph()
 3: while not q.isEmpty() do
 4:   r ← q.remove()
 5:   if r nicht markiert then
 6:     def ← defUse.getDef(r)
 7:     if def ist eine PhiNode then
 8:       phis.add(def)
 9:     else
10:       if def ist vom Typ Labelable then
11:         processInstruction(def)
12:       end if
13:     end if
14:     for use in defUse.getUses(r) do
15:       if use ist eine PhiInstruction then
16:         phis.add(use)
17:       else
18:         if use ist vom Typ Labelable then
19:           processInstruction(use)
20:         end if
21:       end if
22:     end for
23:   end if
24: end while
25: return g
```

Es werden ausgehend von den bereits markierten Referenzen alle verbundenen Instruktionen (die Definition und alle Uses) betrachtet. Diese werden, soweit sie keine `PhiNodes` sind, mittels der Funktion `processInstruction` behandelt. Diese Funktion setzt für den gegebenen Instruktions-Knoten das

aktuell verarbeitete Label, wenn es bisher noch nicht markiert wurde. Wird dieser Knoten markiert, werden über Methode `getLabelableRefs` im Typ `InstructionNode` alle mit diesem Knoten verbundenen Referenzen gesucht. Im Abschnitt 3.2.4 wird diese Methode vorgestellt. Stellt die betrachtete Instruktion eine ϕ-Instruktion dar, wird diese separat betrachtet und wird in einer temporären Sammlung `phis` hinzugefügt. Die Verarbeitung von ϕ-Knoten innerhalb des Graphen wird im Abschnitt 3.2.6 betrachtet.

3.2.4 getLabelableRefs

Bei der Methode im Typ

3.2.5 Umgang mit mehreren Labels

3.2.6 Umgang mit Phi-Knoten

ϕ-Instruktionen innerhalb des Datenflussgraphen erfordern, aufgrund ihrer Eigenheiten, eine besondere Behandlung.

Chapter 4

Transformation

In diesem Kapitel sollen die Überlegungen und der Prozess der Bytecode Transformation, auf Basis der Resultate aus dem Analyse Prozess, vorgestellt werden. Es wird zunächst auf die Beschaffung der nötigen Informationen eingegangen, um im Anschluss die Regeln, nach denen Bytecode generiert oder manipuliert wird, erläutert.

Ziel der Transformation ist es zum Einen an den Grenzen der Bubble Konvertierungen zwischen den Originalen und den Optimalen Typen in den Sourcecode einzufügen. Zum Anderen müssen Uses markierte Uses in entsprechende optimierte Versionen umgewandelt werden.

4.1 Lokale Variablen

4.1.1 Optimierte Variablen

Um originale lokale Variablen im Bytecode nicht mit den optimierten Versionen zu überschreiben wurde eine Abbildung geschaffen, die jeder lokalen Variable ein Tupel zuweist. $l \rightarrow (L, l')$, wobei l die originale lokale, L ein Label und l' die optimierte Variable für das Label L darstellt. So ist sichergestellt, dass optimierte und die entsprechende originale Referenz in zwei verschiedenen Lokalen geführt werden. Darüber hinaus ist diese Trennung wichtig, da die JVM die Plätze für lokale Variablen typisiert und daher nicht an verschiedenen Stellen im Programm verschiedene Typen in derselben lokalen Variable gespeichert werden können.

4.1.2 Variablen zu Value Numbers

Da der IR mit den beinhalteten value numbers eine Abstraktion des eigentlichen Bytecodes darstellt, fehlt auch jeglicher Bezug zu den eigentlichen lokalen Variablen, die von einer spezifischen value number dargestellt wird. Darüber hinaus muss für Definition einer Instruktion das `STORE` (schreibt die auf dem Stack liegende Referenz in die gegebene lokale Variable) und für alle Uses entsprechende `LOAD`s (ließt die gegebene lokale Variable) im Bytecode lokalisiert werden. Diese Informationen sind nötig, da im Falle von Optimierungen, die für die entsprechende Instruktion erzeugt werden, die optimierten statt die originalen Referenzen geladen werden müssen.

Diese Informationen werden in der `InstructionNode` gehalten. Beim erzeugen eines solchen Objekts wird in der `InstructionNodeFactory` zum einen versucht die lokalen zu den verwendeten value numbers zu erschließen und zum anderen die auf Position im Bytecode zu schließen an der die entsprechenden Werte auf den Stack gelegt werden.

Der IR, der aus einer class-Datei erzeugt wird besitzt ein privates Feld `localMap` vom Typ `com.ibm.wala.ssa.IR.SSA2LocalMap`, welche in ihrer einzigen Implementierung (der `com.ibm.wala.ssa.SSABuilder.SSA2LocalMap`) eine private Methode mit Signatur `int[] findLocalsForValueNumber(int, int)`, welche für eine gegebenen Bytecode Index und value number alle möglichen lokalen Variablen für diese value number an der gegebenen Stelle zurückgibt. Um diese Methode trotz aller Zugriffsbeschränkungen aufzurufen wurde eine Methode in *Groovy* geschrieben um auf diese Methode zuzugreifen. Beim Suchen nach lokalen Variablen muss zwischen value numbers als Definition und als Use unterschieden werden. Das folgende Beispiel beschreibt das Problem bei Definitionen:

```
INVOKEVIRTUAL org/example/SomeType.f()I // index 1
ISTORE 5                                // index 2
```

Figure 4.1: Lokale Variable für Definition

Die lokale Variable der Definition der `INVOKEVIRTUAL` Instruktion ist zum Zeitpunkt des Methodenaufrufs noch nicht bekannt. Erst im Index 2 wird dieser Wert der lokalen Variablen 5 zugewiesen.

Um nun die Stellen zu finden an denen Variablen auf den Stack gelegt oder vom Stack gepoppt werden wurde eine einfache Stacksimulation eingeführt,

wie sie in Algorithmus 3 zu sehen ist.

Algorithm 3 Simulation des Stacks

```
1:  size ← Höhe des Stacks zum Zeitpunkt der Instruktion
2:  index ← Index der betroffenen Referenz innerhalb des Stacks
3:  bcIndex ← Bytecode Index der betroffenen Instruktion
4:  while actBlock.getPredNodes() = 1 do
5:    actBlock ← callGraph.getBlockFor(bcIndex)
6:    while bcIndex > actBlock.getFirstInstructionIndex() do
7:      bcIndex ← bcIndex − 1
8:      instruction ← instructions[bcIndex]
9:      if instruction.getPushedCount() > 0 then
10:       size ← size − 1
11:       if index == size then
12:         return bcIndex
13:       end if
14:     end if
15:     size ← size + instruction.getPoppedCount()
16:   end while
17: end while
18: return −1 // kein Index gefunden
```

Dieser Algorithmus funktioniert für Definitionen, also das Suchen von `STORE` Instruktionen, ähnlich. Der Unterschied liegt dabei ausschließlich im Inkrementieren (statt Dekrementieren) der *bcIndex* Variable und dem umgekehrten Verhalten beim *push* bzw. *pop* von Werten auf bzw. vom Stack.

Diese Informationen werden in dem entsprechenden `InstructionNode` Objekt gespeichert. Zu diesem Zweck besitzt dieser Typ drei Abbildungen ($\mathbb{N} \to \mathbb{N}$), die zum Zeitpunkt der Erstellung befüllt werden:

`localMap` bildet eine value number auf eine lokale Variable ab

`loadMap` bildet eine lokale Variable auf einen Bytecode Index ab, an dem diese Variable auf den Stack geladen wurde

`storeMap` bildet eine lokale Variable auf einen Bytecode Index ab, an dem `STORE` diese Referenz in die entsprechende Variable schreibt

4.2 Bytecode Generierung

4.2.1 Informationen des TypeLabels

4.2.2 Konvertierung

4.2.3 Optimierung

Chapter 5

Auswertung

Chapter 6

Fazit