Optimierung mittels der Auswahl von String Repräsentationen in Java Bytecode

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

October 12, 2014

Contents

List of Figures

List of Algorithms

Abstract

Es geht um blibla blubb

Chapter 1

Einleitung

Die Verwendung von optimierten Datentypen beim Entwickeln eines Programms ist schwierig. Um eine Optimierung optimal einzusetzen, muss der Entwickler nicht nur deren API kennen und nutzen. Zusätzlich muss der richtige Einsatzzweck der entsprechenden Optimierung auch bekannt und ebenso verwendet werden. Des weiteren sind ihnen die ursprünglichen Datentypen eher vertraut als diejenigen Typen, die für ein spezielles Problem entworfen worden sind. Daraus folgend ergibt sich das Problem spezielle Optimierungen richtig im Quellcode zu verwenden.

Ähnliches gilt für den Code von bestehenden Altsystemen. Dabei ergibt sich allein durch die Menge an Code die refaktorisiert werden müsste um eine oder mehrere Optimierungen einzufügen, ein nicht unerheblicher Aufwand.

Ziel dieser Arbeit ist es daher zu untersuchen, ob sich Programme durch Auswahl und Substitution von alternativen Datentypen automatisiert optimieren lassen. Diese Untersuchung kann allerdings keine allumfassende Auswertung bereitstellen und wird sich auf die Optimierung der Laufzeit des `java.lang.String` Datentyps der *Java* Plattform und der Programmiersprache *Java* beschränken. Nach der Auswertung soll eine Aussagedarüber möglich sein, ob diese Optimierungen möglich sind.

Im Rahmen dieser Arbeit soll ein System erstellt werden, dass String Operationen in einem gegebenen Java Programm mit einer entsprechenden optimierten Version auf Basis des Java Bytecodes ersetzt. Dabei soll das System anhand statischer Code Analyse die Stellen lokalisieren, an denen eine Optimierung angewendet werden kann, und mittels der Transformation des Bytecodes des Programms diese Optimierungen anwenden. Als Ergebnis wird ein lauffähiges Programm erwartet, dass bei gleicher Eingabe eine

geringere Ausführungszeit besitzt als das originale Programm.

In Kapitel 2 werden die Werkzeuge und Grundlagen beschrieben, auf denen das in dieser Arbeit entwickelte System aufbaut. Kapitel 3 erläutert den Analyse Prozess des Systems, indem die Datenstrukturen sowie der Algorithmus für die statische Code Analyse beschrieben wird. In Kapitel 4 wird basierend auf den Analyse Ergebnissen die Transformation des Bytecodes beschrieben. Kapitel 5 widmet sich der Auswertung der Tests in Form von Benchmarks und begründet diese Ergebnisse. Kapitel 6 zieht ein Fazit und beschreibt mögliche Weiterentwicklungen des Systems.

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 in dieser Arbeit 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 `DataFlowGraphs` 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. Die verwendete Queue Implementierung ist eine auf einem `java.util.LinkedHashSet` basierende Eigenentwicklung mit dem Namen `de.unifrankfurt.faststring.analysis.util.UniqueQueue`.

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.

optimizedParams Ein Objekt. Dabei ist jeder key die ID einer Methode und der entsprechende value eine Parameter Liste in interner JVM Notation. Wird bei einer Optimierung einer Methode eine andere Parameter Liste erwartet als die ursprüngliche der originalen Methode, so muss die neue Signatur an dieser Stelle angegeben werden.

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.

3.1.3 Konventionen

Bei dem Erstellen von optimierten Typen sind einige Konventionen zu befolgen. Das System rechnet damit, dass diese Annahmen befolgt werden. Diese Regeln sind im Folgenden beschrieben.

Methodennamen

Wird eine Methode als *effectedMethod* deklariert, wird der optimierte Gegensatz durch den Namen identifiziert. Wenn z.B. die Methode `f()` aus dem Typ `A` optimiert werden soll, so muss in dem optimierten Typ `AOpt` eine Methode `f()` existieren. Besitzt die originale Methode eine Parameterliste, so muss diese mit derer der optimierten Methode in Anzahl und Typen übereinstimmen. Eine Ausnahme bildet dabei die Verwendung von ebenfalls optimierten Parametern. In diesem Fall muss im Feld *optimizedParams* in der *type* Datei ein entsprechender Eintrag erfolgen.

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. Zum Erstellen des Graphen werden wie in 3.1.1 beschrieben zunächst initiale Referenzen benötigt. Diese Referenzen werden über die Methode `Set<Reference> findTypeUses(AnalyzedMethod)` ermittelt, welche sich in der abstrakten Klasse `BaseTypeLabel` befindet. Diese Methode ist auch Teil des `TypeLabel` Interfaces. Grundlage dieser Ermittlung sind die von der in der Label Definition definierten *effectedMethods*, also derjenigen Methoden für die dieses Label eine Optimierung darstellt. Die `findTypeUses` Methode erstellt für jede Referenz, auf der in der gegebenen Methode eine der *effectedMethods* aufgerufen wird, ein `Reference` Objekt. Dabei wird für jede value number genau ein Referenz Objekt erzeugt. Die erstellten Referenzen werden dem verwendeten Label markiert.

Die Implementierung des Algorithmus befindet sich als statische Methode in der Klasse `LabelAnalyzer`. Ausgehend von den initial markierten Knoten, wird jeweils die Definition sowie alle Uses einer Referenz betrachtet, ob diese mit dem Label der Referenz markiert werden können. Der Algorithmus 2

beschreibt die Implementierung dieser Vererbung des Labels. Als Queue findet wieder die `de.unifrankfurt.faststring.analysis.util.UniqueQueue` Verwendung.

Algorithm 2 Vererbung des Labels

```
 1: q ← new Queue(initialReferences)
 2: g ← der übergebene Datenflussgraph
 3: while not q.isEmpty() do
 4:   r ← q.remove()
 5:   if r.label is not null then
 6:     def ← r.getDef()
 7:     if def.canProduce(r.label) then
 8:       labelConnectedRefs(def)
 9:     end if
10:     for use in r.getUses() do
11:       if use.canUseAt(r.label, useIndex) then
12:         labelConnectedRefs(use)
13:       end if
14:     end for
15:   end if
16: end while
17: return g
```

Eine wichtige Rolle während des Vererbungsprozesses spielen dabei die Methoden `boolean Labelable.canProduce(TypeLabel)` und `boolean Labelable.canUseAt(TypeLabel, int)`. Diese beiden Methoden bestimmen für eine Referenz ob ein Label auf eine Definition oder eine Benutzung vererbt werden kann.

`canProduce` sagt aus ob eine Instruktion eine optimierte Referenz definieren kann. Die Implementierungen der `canProduce` Methode sehen für die einzelnen `Labelable` Subtypen wie folgt aus:

NewNode gibt immer `true` zurück

ConditionalBranchNode gibt immer `false` zurück

MethodCallNode Delegation an die Methode `canBeDefinedAsResultOf` des übergebenen Label Objekts.

Die Methode `canUseAt` betrifft dagegen nur Benutzungen von Referenzen. Die Semantik hinter der Methode ist ob diese Instruktion eine optimierte Referenz verwenden kann. Der Index stellt dabei den Parameter Index das an dem diese Referenz in dieser Instruktion verwendet wird. Die Implementierung dieser Methode sieht für die einzelnen `Labelable` Subtypen wie folgt aus:

NewNode gibt immer `false` zurück

ConditionalBranchNode gibt immer `true` zurück

MethodCallNode Wenn die Methode nicht statisch und der Index 0 ist, wird an die Methode `canBeUsedAsReceiverFor` delegiert, andernfalls an die Methode `canBeUsedAsParamFor`.

Ist für eine Instruktion die Prüfung für die entsprechende Methode positiv, wird diese Instruktion an die Hilfsfunktion `labelConnectedRefs` weitergeleitet. Diese Methode ist im folgenden dargestellt:

Algorithm 3 labelConnectedRefs

```
1: node.setLabel(label)
2: for ref in node.getLabelableRefs() do
3:   if ref.getLabel() is not null then
4:     ref.setLabel(label)
5:     q.add(ref)
6:   end if
7: end for
```

Wobei sowohl die Queue q, als auch das betreffende TypeLabel *label* aus dem umgebenen Kontext im Algorithmus 2 auch in dieser Funktion zur Verfügung stehen.

3.2.4 Umgang mit mehreren Labels

Es ist möglich für die Analyse mehrere Labels für die Analyse zu verwenden. Diese lassen sich dem `MethodAnalyzer` als `Collection` im Konstruktor zusammen mit der zu analysierenden Methode übergeben. Vor der Analyse werden zuerst alle Referenzen identifiziert von denen eine zu optimierende

Methode verwendet wird. Diese Referenzen werden zu einer Menge an Referenzen vereint. Die auf diese Weise erzeugten Referenzen besitzen wie in 3.2.3 beschrieben jeweils das Label für das die Referenz erzeugt wurde.

Ein Problem ergibt sich, stellt man der Analyse mehrere Labels zur Verfügung, die allerdings Optimierungen für dieselbe Methode des selben Typs beschreiben. Die erzeugten initial erzeugten Referenz-Objekte, welche die zu optimierende Methode verwenden, werden bei ihrer Erzeugung mit dem entsprechenden Label markiert. Dieses Implementierungsdetail führt dazu, dass die entsprechenden Referenzen mit demjenigen Label markiert werden, welches zuerst verarbeitet wird. Dieses wird immer das Label sein, welches einen geringeren Index in der Liste von Labels besitzt.

Stellt man der Methode nun zwei unterschiedliche Labels zur Verfügung, die allerdings den selben originalen Typen verwenden, ergibt sich ein anderes Problem, wenn innerhalb der analysierten Methode auf einer Referenz sowohl die eine als auch die andere Methode aufgerufen wird. Da das System jeder Referenz in einer Methode nur ein Label zuweisen kann, kann diese Referenz, welche beide Methoden der beiden TypeLabels verwendet, nur mit einem der TypeLabels markiert werden. In diesem Szenario wird wieder dasjenige Label für die Markierung verwendet, welches den niedrigeren Index innerhalb der Liste besitzt.

Nachdem der Datenflussgraph initial erzeugt wurde, wird der Algorithmus wie in 3.2.3 beschrieben ausgeführt. Da Labels nur auf Knoten vererbt werden, gilt dass der entsprechende Knoten dasjenige Label erhält, welches zuerst auf diesen Knoten vererbt wird.

3.2.5 Phi-Knoten

ϕ-Instruktionen innerhalb des Datenflussgraphen erfordern, aufgrund ihrer Eigenheiten, eine besondere Behandlung. Da sie keine Instruktionen im eigentlichen Sinne darstellen, sondern nur ein Zeiger alias für andere Referenzen darstellen, lassen sich diese Knoten auch nicht optimieren. Obwohl sie selber nicht optimiert werden können, so lässt sich doch über diese Knoten hinweg ein Label vererben. Allerdings sollte unterschieden werden zwischen denjenigen verbundenen Referenzen, für die eine Markierung tatsächlich sinnvoll und für welche eher weniger sinnvoll ist.

Daher sollten diejenigen Referenzen markiert werden, für die auch Optimierungspotenzial besteht. Wohingegen diejenigen Referenzen, für die keinerlei Optimierung notwendig ist, auch keine Markierung erhalten sollten.

Wenn das System beim Vererben der Markierung einen ϕ-Knoten als Instruktion verarbeitet, wird dieser auf den ϕ Stapel gelegt und alle verbundenen Referenzen für die weitere Verarbeitung vorgemerkt. Nachdem alle Referenzen verarbeitet wurden, wird dieser Stapel durchlaufen. Für jede `PhiInstructionNode` in diesem Stapel wird entschieden ob und mit welchem Label diese Instruktion markiert wird.

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 optimierten 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 4 zu sehen ist.

Algorithm 4 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

Die Generierung von neuem und die Manipulation von bestehendem Bytecode wird von einer Instanz der Klasse `MethodTransformer` vorgenommen. Als Eingabe dient eine Instanz vom Typ `TransformationInfo`, welche auf einem `AnalysisResult` basiert und Informationen über die verwendeten lokalen Variablen zur Verfügung stellt. Diese Informationen sind zum einen die verwendeten Variablen, als auch die Abbildung von originalen Lokalen zu optimierten.

Anhand dieser Informationen werden sowohl Konvertierungen als auch Optimierungen zu dem bestehenden Bytecode hinzugefügt. Eine Konvertierung beschreibt dabei eine Transformation von einem originalen zu einem optimierten Typ bzw. von einem optimierten Typ zu einem originalen Typ. Eine Optimierung hingegen ersetzt eine Instruktion mit einer optimierten Variante. Diese Mechanismen sollen in dem folgenden Abschnitt vorgestellt werden.

4.2.1 Informationen des TypeLabels

Um Optimierungen und Konvertierungen auf bestimmten Typen umzusetzen werden verschiedene Informationen benötigt. Diese sind:

- Wie der optimierte Type heißt
- Wie ein optimierter Typ aus einem Originalen erzeugt wird
- Wie die Signatur einer optimierten Variante einer Methode aussieht

Diese Informationen werden über die `TypeLabel` Definitionen dem System zur Verfügung gestellt. Zu diesem Zweck enthält das `TypeLabel` Interface, zusätzlich zu denen in 3.1.2 vorgestellten, die folgenden Methoden:

`getOptimizedType()` Gibt ein `Class` Objekt des optimierten Typs zurück

`getOriginalType()` Gibt ein `Class` Objekt des originalen Typs zurück

`getCreationMethodName()` Definiert den Namen der statischen Methode innerhalb des optimierten Typs, die ein neues Objekt des optimierten Typs erzeugt

`getToOriginalMethodName()` Definiert den Namen der Methode, die auf einem Objekt des optimierten Typen aufgerufen werden kann um eine äquivalente Instanz des originalen Typen zu erzeugen

`getReturnType(MethodReference)` gibt den Rückgabe Typ der gegebenen Methode im optimierten Typ zurück. Der Typ wird als `String` Repräsentation in der internen JVM Form zurück gegeben. (Beispiel: `Ljava/lang/String;`)

`getParams(MethodReference` gibt die Parameter der gegebenen Methode im optimierten Typ zurück. Die Parameter werden als `String` konkatenierte Liste umgeben mit Klammern zurück gegeben. Die Typen der Parameter werden als interne JVM Form angegeben (Beispiel: (`II`)).

4.2.2 Konvertierung

Konvertierungen dienen der Kompatibilität zwischen Code innerhalb und außerhalb der "Bubble". Dabei betreffen diese ausschließlich Referenzen im Datenflussgraphen. Formal lässt sich dieser Vorgang wie folgt beschreiben. Die Funktion $label : E \to L$, wobei L die Menge aller in dem System vorhandenen Label ist und $E = R \cup I$, weißt jedem Knoten eine Label Markierung zu. Ein Knoten ohne Markierung bekommt das Label $KEIN\_LABEL$ zugewiesen. Bei der Entscheidung ob zwischen einer Instruktion i und einer Referenz r eine Konvertierung nötig ist sind zwei Entscheidungen zu treffen:

1. Handelt es sich bei der Instruktion um eine *labelable* Instruktion
2. Ist es eine Def ($i \to r$) oder eine Use ($r \to i$) Kante innerhalb des Graphen

Ist die betrachtete Instruktion ohnehin nicht in der Lage mit einem Label markiert zu werden, so muss nur die Referenz betrachtet werden und wenn $label(r) \neq KEIN\_LABEL$, dann muss eine Konvertierung an dieser Kante eingeführt werden.

Handelt es sich bei der Instruktion allerdings um eine *labelable* Instruktion, so trifft die Methode `needsConversationTo(Label)` (s. Algorithmus 5) des Objekts auf den die Kante zeigt die Entscheidung ob eine Konvertierung nötig ist.

Algorithm 5 needsConversationTo(Label)

```
1: if isSameLabel(label) then
2:   return false
3: end if
4: if label ≠ null then
5:   return not label.compatibleWith(this.label)
6: else
7:   return not this.label.compatibleWith(null)
8: end if
```

Konvertierungen werden von Instanzen vom Typ `Converter` durchgeführt. Dabei wird zwischen Definitions- und Use-Konvertierungen unterschieden für jeweils eigene Implementierungen der `Converter` Schnittstelle existieren. Der `Converter` nutzt dabei das *Visitor* Muster, welches für die einzelnen Instruktionsknoten implementiert wurde. Die jeweiligen *visit*-Methoden erstellen auf der zur Verfügung gestellten `ConversionPatchFactory` die Shrike Patch Objekte um die Bytecode Manipulationen durchzuführen.

Die `ConversionPatchFactory` dient der Erstellung von Shrike Patch Objekten und dem Hinzufügen dieser in den Bytecode der aktuell verarbeiteten Methode. Zusätzlich muss bei der Instanziierung eines Objektes angegeben für welche Konvertierung diese Factory verwendet wird. Zu diesem Zweck besitzt der Typ zwei Attribute vom Typ `TypeLabel`:

`from` Der Typ den ein Objekt vor der Konvertierung besitzt

`to` Der Typ in den ein Objekt konvertiert werden soll

Die Methoden, die zum Erzeugen eines Patches zur Verfügung stehen, sind die Folgenden:

`createConversationAtStart(local)` Erstellt eine Konvertierung am Anfang der Methode für die gegebene lokale Variable

`createConversationAfter(local, bcIndex)` Erstellt eine Konvertierung nach dem gegebenen Bytecode Index für die gegebene lokale Variable. Die Variablen Angabe kann auch weggelassen werden, dann wird der konvertierte Wert nicht in die entsprechende optimierte Variable gespeichert.

`createConversationBefore(local, bcIndex)` Erstellt eine Konvertierung vor dem gegebenen Bytecode Index für die gegebene lokale Variable. Die Variablen Angabe kann auch weggelassen werden, dann wird der konvertierte Wert nicht in die entsprechende optimierte Variable gespeichert.

Soll eine Konvertierung für eine Definition ohne eine gegebene lokale Variable durchgeführt werden, so wird ausschließlich Methodenaufruf zur Konvertierung zum bzw. vom optimierten Typ aufgerufen. Wird eine lokale Variable angegeben so wird zunächst die Referenz auf dem Stack mittels der `DUP` Anweisung verdoppelt. Nach dem Aufruf der Konvertierung Methode wird die optimierte Referenz in die entsprechende optimierte Variable gespeichert. Konvertierungen die am Anfang der Methode durchgeführt werden (was ausschließlich für Konvertierungen die Übergabeparameter betreffen) verhalten sich genauso wie Konvertierungen denen eine Variable mitgegeben wurde, mit dem Unterschied, dass anstatt der `DUP` Anweisung ein `LOAD` für die entsprechende Variable verwendet wird um die Referenz für den Methodenaufruf zu verwenden.

Wird eine Konvertierung für einen Use benötigt, muss die originale Referenz bevor sie verwendet wird, wieder in den originalen Type umgewandelt werden. Daher wird an der Stelle nachdem die betroffene Referenz auf den Stack gelegt wurde der Methodenaufruf zum Umwandeln in den originalen Typ eingefügt. Ist die lokale Variable allerdings bekannt, wird die `LOAD` Instruktion, die die originale Variable lädt ersetzt durch ein Laden der optimierten Variable um diese danach in einen originalen Typ zu konvertieren.

4.2.3 Optimierung

Für jede *labelable* Node wird, wenn sie mit einem Label markiert ist, eine Optimierung durchgeführt. Dafür sind 3 Schritte notwendig, diese werden im Folgenden beschrieben.

Primäres Ziel der Optimierungstransformation ist es, die originalen Methodenaufrufe durch die Optimierten zu ersetzen. Wie in 3.1.3 beschrieben, wird für eine Methoden Optimierung immer derselbe Name angenommen. In jedem Fall wird zunächst der Typ des Empfänger Objekts auf den des optimierten Typ gesetzt. Ist für die Parameter Liste im optimierten Typ keine Alternative angegeben, wird diejenige der originalen Methode verwendet. Der Rückgabe Typ wird ebenfalls durch das *Typelabel* bestimmt. So

wird das Ziel der `INVOKE` Instruktion auf den optimierten Typ gesetzt.

Um die Methode auf der optimierten Referenz aufrufen zu können, muss diese, statt der originalen Referenz, zum Zeitpunkt des Methoden Aufrufs auf dem Stack vorhanden sein. Darüber hinaus müssen, wenn es sich bei den Parametern um gelabelte Referenzen handelt, auch die optimierten Parameter Referenzen, statt der originalen, auf dem Stack liegen. Daher müssen für optimierte Referenzen die `LOAD` Anweisungen, die die Referenzen für den Methodenaufruf auf den Stack legen, die optimierte statt der originalen Variablen laden.

Ist der Rückgabewert eines optimierten Methodenaufrufs ebenfalls markiert und wird mittels einer `STORE` Anweisung in eine Variable gespeichert, so muss diese Anweisung die optimierte Referenz in die entsprechende optimierte Variable speichern. Daher wird die entsprechende Anweisung (wenn die Methode als *producing* definiert ist) ersetzt durch eine `STORE` Anweisung, die die zurückgegebene Referenz in die entsprechende optimierte Variable speichert.

4.3 Schwierigkeiten

4.3.1 Innere Archive (JARs)

Um Klassen eines gegebenen Programms zu lesen und zu verarbeiten bietet WALAs Shrike die Klasse `OfflineInstrumenter` an. Diese Klasse erwartet eine JAR Datei und ermöglicht daraufhin eine Iteration über alle Klassen innerhalb dieses Archivs.

Befinden sich nun Klassen dieser Anwendung verpackt in einem Archivs innerhalb dieses Archivs, ist es dem `OfflineInstrumenter` nicht möglich diese Klassen zu finden und damit auch zu verarbeiten. Daher ist das System nicht in der Lage Klassen die nicht innerhalb des zu verarbeitenden Java-Archivs zu finden sind zu verarbeiten.

4.3.2 Typisierung von generiertem Bytecode

Bei dem Erzeugen von Bytecode mit Shrike

Chapter 5

Auswertung

In diesem Kapitel soll das entwickelte System einer Auswertung unterzogen werden. Im ersten Abschnitt soll zunächst das das verwendete Werkzeug *JMH* sowie die allgemeine Durchführung der Tests beschrieben werden. Im zweiten Abschnitt folgen dann die einzelnen Tests sowie deren Auswertungen.

5.1 Vorraussetzungen

5.1.1 Java Microbenchmarking Harness

5.1.2 Test Durchführungen

5.2 Benchmarks

5.2.1 Test 1

5.2.2 Test 2

5.2.3 Test 3

5.2.4 Test 4

Chapter 6

Fazit