Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Visualisieren von Algorithmen in Compilern

Tom Schreiner

Bachelorarbeit

Beginn der Arbeit:	05. November 2024
Abgabe der Arbeit:	05. Februar 2025
Gutachter:	John Witulski
	Fabian Ruhland

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 05. Februar 2025

Tom Schreiner

Zusammenfassung

Die Ziele dieser Bachelorarbeit waren einerseits einige Algorithmen und Konzepte aus dem Compilerbau zu visualisieren, aber auch ein Framework zu entwickeln welches diese und beliebige weitere Algorithmen verständlich und Schritt für Schritt visualisieren kann.

Folgende Algorithmen wurden implementiert:

1. Analyse von erreichenden Definitionen für Grundblöcke
2. Liveness Analyse für Grundblöcke
3. Liveness Analyse für einzelne 3-Address-Code Instruktionen
4. Erstellen von Grundblöcken für ein Programm geschrieben in 3-Address-Code
5. Erstellen eines Kontrollflussgraphen für ein 3-Address-Code Programm

Um diese Algorithmen simpel und gut verständlich zu visualisieren brauchte das Framework zwei Arten von Darstellungen:

1. Graphen: um Kontrollflussgraphen darzustellen
2. Tabellen: um Datenflusswerte darzustellen und 3-Address-Code in einer angenehmen Art und Weise zu visualisieren

Desweiteren brauchte es die Möglichkeit 3-Address-Code und Grundblöcke einfach zu verarbeiten.

Außerdem ist es durch Implementierung eines Interfaces einfach weitere Algorithmen hinzuzufügen. Es wurden weitere Interfaces implementiert mit denen häufig genutzte Funktionalitäten wie zum Beispiel einen Button, um Code aus einer Datei zu laden, einfach in neuen Plugins genutzt werden können. Somit können weitere Plugins mit sehr viel weniger Aufwand hinzugefügt werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Als ich im Wintersemester das Modul "Compilerbau"belegt habe, habe ich mich das erste mal mit einigen Algorithmen beschäftigt die eben in diesem Themengebiet angewendet werden. Dabei fiel es mir bei einigen schwer mir diese ohne weiteres vorzustellen.

In dieser Bachelorarbeit wird ein Framework in Java entwickelt, mit dem es einfach sein soll Algorithmen zu visualisieren. Zudem werden einige der Algorithmen, die im Compilerbau verwendet werden, implementiert.

Das Framework basiert darauf, dass durch Implementierung von Interfaces einfach neue "Plugins"hinzugefügt werden können.

1.2 Theoretische Grundlagen

Im folgenden Abschnitt werden die für diese Arbeit notwendigen grundlegenden Konzepte und Algorihmen erklärt da im weiteren Verlauf der Arbeit nur auf die Implementierung dieser eingegangen wird.

1.2.1 Drei Address Code

Drei Address Code ist eine Art von Zwischencode. Charakterisierend für Drei Address Code(folgend auch 3AC genannt) ist, dass einzelne Instruktionen auf maximal drei Addressen(beziehungsweise Variablen oder Konstanten) zugreifen, also die Variable in die der resultierende Wert gespeichert wird und eine oder zwei Variablen oder Konstanten aus denen sich der Resultierende Wert bildet. Dazu ist noch die Operation die ausgeführt wird angegeben.

Für diese Bachelorarbeit wurde eine Teilmenge des im Drachenbuch[Aho08] bescriebenen 3AC verwendet.
Folgende Operationen gibt es:

1. Binäre Operationen X = Y op Z
   In denen das Resultat aus einer der folgenden binären Operation in der Addresse X gespeichert wird. Implementiert wurden Addition, Subtraktion, Multiplikation und Division.

2. Die unäre Operation X = - Y
   In der der invertierte Wert von Y in X gespeichert wird.
3. Der Kopierbefehl X = Y
   In der der Wert Y in X kopiert wird
4. Der unbedingte Sprung goto X
   Hier wird kein Wert gespeichert, sondern zu der Addresse die in X gespeichert ist gesprungen
5. Die bedingten Sprünge if Y goto X und ifFalse Y goto X
   In denen wenn der Wert Y entweder true oder false repräsentiert zur Addresse X gesprungen wird, oder die nächste Instruktion ausgeführt wird wenn dies nicht der Fall ist.
6. Die bedingten Sprünge if Y relOp Z goto X
   In denen auch zu X gesprungen wird, wenn die Relation Y relOp Z wahr ist, sonst wird auch hier die nächste Instruktion ausgeführt.
   Die Implementierte Relationen sind: Y kleiner als Z, Y kleiner oder gleich Z, Y größer als Z, Y größer oder gleich Z, Y ist gleich Z und Y ist ungleich Z.

Hierbei können die Addressen X, Y und Z beliebige Zeichenfolgen sein, Y und Z können ausserdem Konstanten sein. Da sich die Algorithmen in dieser Arbeit nicht mit komplexeren Aufgaben wie Speichermanagement beschäftigen, wurde sich dagegen entschieden den 3AC umfangreichen zu Modellieren.

1.2.2 Grundblöcke

Grundblöcke sind Partitionen eines Zwischencodeprogrammes welche immer zusammen ausgeführt werden.[Aho08] Dies ermöglicht es, einen Block an Instruktionen anzuschauen und bestimmte Optimierungsalgorithmen auf sie anzuwenden, ohne Gefahr zu laufen die Semantik des Programmes zu verändern.
Um ein Programm in Grundblöcke aufzuteilen kann man wie folgt vorgehen[Aho08]:

1. Markiere die erste Instruktion des Programmes, da diese immer ausgeführt wird.
2. Markiere alle Instruktionen die Ziel eines Sprungs sind oder auf einen Sprung folgen.
3. Alle Instruktionen die auf eine markierte Instruktion folgen, bis zu einer neuen markierten Instruktion gelten nun als ein Grundblock.

Folglich kann man für jeden Grundblock bestimmen, welche Grundblöcke auf ihn folgen, daraus lässt sich ein Flussgraph bestimmen den wir, da er den Kontrollfluss beschreibt, folglich Kontrollflussgraphen nennen werden. In diesem stellt jeder Grundblock

einen Knoten dar und jede Kante einen Sprung von einem Grundblock zum anderen.

1.2.3 Kontrollflussgraphen

Gehen wir genauer auf Kontrollflussgraphen ein:
Diese sind gerichtete Graphen, deren Knoten aus Grundblöcken und deren Kanten aus den jeweilig folgenden Grundblöcken besteht.
Auf ihnen lassen sich bestimmte Mengen für alle Grundblöcke definieren, welche für die in dieser Arbeit implementierten Datenflussanalysen benötigt werden:

1. Die $gen[B]$ Menge:
   Beschreibt alle Instruktionen in einem Grundblock B welche einer Addresse einen Wert zuweisen.
2. Die $kill[B]$ Menge:
   Beschreibt alle Instruktionen ausserhalb des Grundblocks B welche einer Addresse einen Wert zuweisen, der durch eine Instruktion in der $gen[b]$ Menge überschrieben wird.
3. Die $def[B]$ Menge:
   In der $def[B]$ Menge sind alle Variablen enthalten, denen im Gundblock B ein Wert zugewiesen wird, bevor diese "verwendet"wird.
4. die $use[B]$ Menge:
   Die $use[B]$ Menge definiert alle Variablen, deren Werte vor ihrer Definition verwendet werden.

1.2.4 Erreichende Definitionen

Dies ist eine der gebräuchlichsten und nützlichsten Datenflussanalysen[Aho08, S.734].
Über sie können wir herausfinden welchen Variablen zu einem bestimmten Zeitpunkt ein Wert zugewiesen ist. Eine Anwendung wäre zum Beispiel zu kontrollieren ob eine Variable zu einem bestimmten Zeitpunkt überhaupt einen Wert hat, sofern die ursprüngliche Programmiersprache dies als notwendig erachtet. Man kann aber auch schauen, ob die Variable eine Konstante ist und somit Instruktionen gespart werden könnten.

1.2.5 Lebendige Variablen

Bei der Analyse lebendiger Variablen(folgend auch liveness Analyse genannt) bringen wir in Erfahrung ob ein bestimmter Wert zu einem bestimmten Zeitpunkt lebendig ist. Leben-

dig bedeutet in diesem Kontext, das dieser Wert definiert wurde und zu einem späteren Zeitpunkt im Programm auch noch genutzt wird.

Die analyse lebendiger Variablen hat viele Anwendungsgebiete, zum Beispiel hat ein reeller Computer nur eine begrenzte Anzahl an Registern, somit können nicht unendlich viele Variablen gleichzeitig zur Benutzung zur Verfügung stehen. Die liveness Analyse kann hier berechnen wie viele Register wir maximal benötigen, da eventuell nicht alle Variablen gleichzeitig lebendig sind, also gebraucht werden. (Frei nach dem Drachenbuch[Aho08] zitiert.)

2 Framework

In diesem Abschnitt soll es um das Implementierte Framework gehen. Aus den Theoretischen Grundlagen lässt sich schließen, dass folgende Daten(-Strukturen) unbedingt im Framework implementiert sein sollten:

1. Drei Address Code Instruktionen:

2. Drei Address Code:

3. Grundblöcke:

Da der Usecase dises Frameworkes ist Algorithmen zu visualisieren braucht es folglich eine Möglichkeit den Code, auf den die Algorithmern laufen, zu visualisieren. Hier wurde sich für eine Tabelle entschieden, da eine 3AC Instruktion in bis zu sechs [1] atomare Zellen aufgeteilt werden kann. Da es sich bei einem Programm immer um eine Liste an Instruktionen handelt, ergibt sich so ein 2-Dimensionales Feld an Daten. Ausserdem möchten wir ja auch die Daten die wir aus unseren Analysen erheben angezeigt bekommen. Auch hier ist eine Tabelle sinnvoll, zum Beispiel um Datenflussmengen unserer Grundblöcke anzuzeigen.

Ein weiterer Algorithmus der implementiert werden soll ist die Erstellung eines Kontrollflussgraphen. Hier ist es natürlich auch wichtig dass dieser dargestellt werden kann, daher sollen auch Graphen dargestellt werden können.

[1] $if\ Y\ relOp\ X\ goto\ L$ ist die längste legale 3AC Instruktion

Aus diesen Anforderungen ergibt sich folgendes Klassendiagramm:

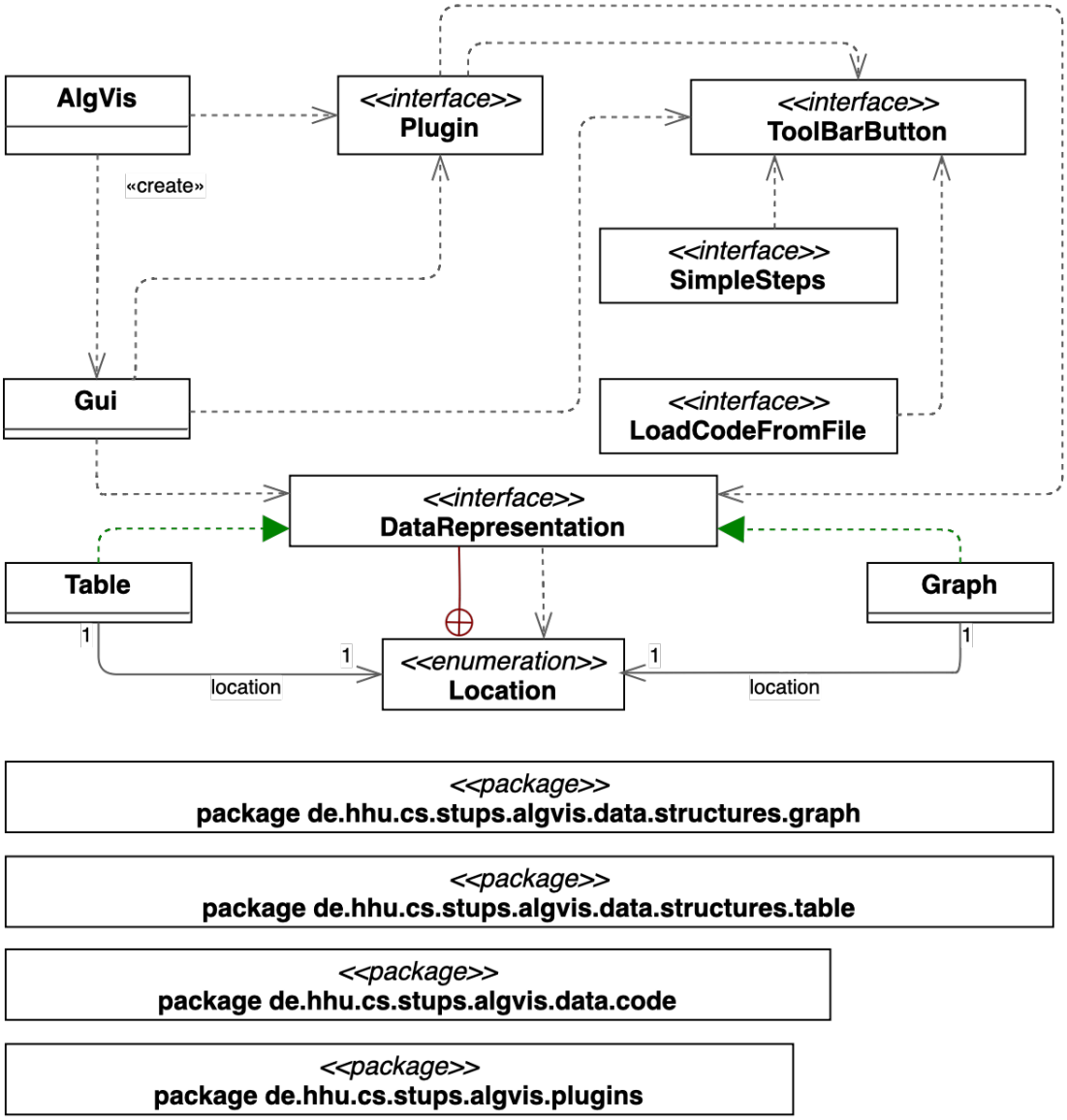

Abbildung 1: Klassenstruktur der GUI-Klassen

Die Klasse $AlgVis$ gilt hier als Entrypoint. Sie initialisiert die Klasse Gui und verwaltet welche Plugins geladen werden.
Das Interface $Plugin$ definiert welche Methoden neue Plugins beziehungsweise Algorithmen benötigen um hinzugefügt zu werden, diese werden in $de.hhu.cs.stups.algvis.plugins$ implementiert. Zudem definiert das Interface $ToolBarButton$ wie Buttons der ToolBar, welche im nächsten Kapitel im Detail erklärt wird, implementiert werden können. Bei-

spiele dafür liefern das $SimpleSteps$ und $loadCodeFromFile$ Plugin, welche vordefinierte Buttons anbieten. Die Packages endend auf $graph$ und $table$ enthalten hierbei Helferklassen für die jeweiligen Komponenten. Im Package $de.hhu.cs.stups.algvis.data.code$ sind die vorhin genannten Datenstrukturen für drei Address Code, Grundblöcke und drei Address Code Instruktionen implementiert. Hierzu auch später mehr.

2.1 Gui

Die Gui Klasse ist das Herzstück der Visualisierung. Hier wird ein $JFrame$, also ein GUI Fenster der Java Standardlibrary $Swing$ geladen. In ihr befinden sich drei GUI-Elemente auch aus der $Swing$-Library:

1. Eine Menüleiste in der über ein Dropdown alle Plugins aufgerufen werden können.
2. Ein $JPanel$ welches $ContentPanel$ genannt wurde, in ihm befinden werden alle grafischen Elemente des aktuell genutzten Plugins geladen. Die verfügbaren grafischen Elemente sind Implementationen des Interfaces $DataRepresentation$, also entweder $Table$ oder $Graph$ auf die an einem späteren Zeitpunkt genauer eingegangen wird. Sollte später ein Plugin eine andere Art der Darstellung benötigen, kann dieses das Interface mit einer anderen awt-Komponente implementieren.
3. Und eine $JToolBar$.
   Da die meißten Plugins ähnliche Funktionalitäten haben, wie zum Beispiel das schrittweise Durchlaufen eines Algorithmuses, wurde ein GUI-Element hinzugefügt in dem Buttons zur Kontrolle des Plugins hinzugefügt werden können.

2.2 Graphen

Graphen werden durch die Klasse $Graph$ und zwei Subklassen $Edge$ und $Node$ realisiert. Da das realisieren einer eigenen Engine für die Darstellung von Graphen diese Bachelorarbeit übertreffen würde wurde entschieden eine externe Library zu verwenden. Hier wurde sich für $GraphStream$ [2] entschieden.

2.3 Tabellen

Um 3AC und ermittelte Datenflusswerte darzustellen wurde ausserdem die Möglichkeit implementiert, Daten als Tabellen darzustellen. Diese ist in der gleichnamigen Klasse $Table$ implementiert und erweitert die $swing$ Komponente $JTable$.

[2] `https://graphstream-project.org`

2.4 Drei Address Code

Im folgenden Kapitel erläutere ich die Implementierung des drei Address Codes. Wie zu Beginn von Kapitel 2 angeführt werden diese Klassen benötigt:

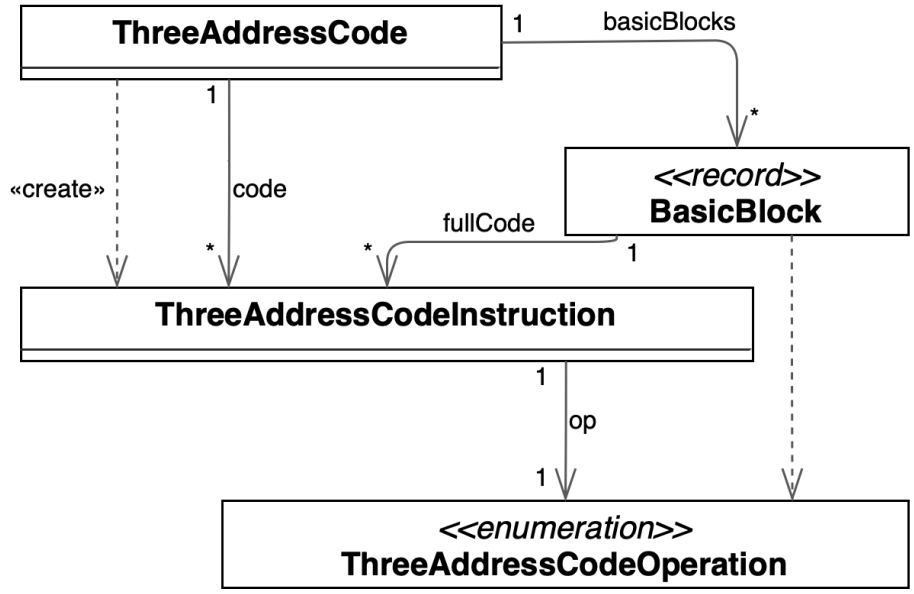

Abbildung 2: Implementation der drei Address Code Klassen

2.4.1 Die ThreeAddressCodeInstruction Klasse

Um einzelne Instruktionen zu modellieren wird die Klasse $ThreeAddressCodeInstruction$ verwendet. Generiert werden diese aus einem $String$ und einer Zahl, nämlich der Addresse an der Stelle ihres Programms. Dies ist zwar streng gesehen eine Ungenauigkeit der Modellierung, da jede Instruktion zwar eine Addresse hat, diese jedoch nicht in Bezug direkt auf die einzelne Instruktion steht, sondern nur im Kontext des Gesamtprogrammes gesehen werden kann. Da es jedoch die Implementierung einiger folgender Methoden stark vereinfacht und um aktuellen Usecase des Frameworks keine Instruktion ohne eindeutige Addresse existiert wurde sich dazu entschieden für diese ein Attribut anzulegen.

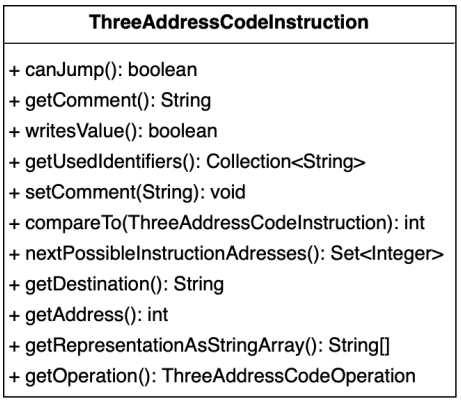

Abbildung 3: Die ThreeAddressCodeInstruction Klasse

Um die in Kapitel 1.2.1 definierten Instruktionen zu modellieren werden folgende weitere Attribute benötigt:

1. Eine $ThreeAddressCodeOperation$ dies ist ein enum, welches definiert welche Operation die angegebene Instruktion ausführt.
2. Ein $String$ namens destination. Dieser gibt entweder das Ziel eines Sprunges an, oder den Ort an dem eine Berechnung gespeichert werden soll.
3. Den $String$ source. Da alle Instruktionen ausser dem unbedingten Sprung mindestens einen Wert verarbeiten.
4. und einen $String$ modifier, für Instruktionen die zwei Werte entweder vergleichen oder arithmetisch verrechnen.

Desweiteren wurde ein weiteres $String$-Attribut zum hinzufügen von Kommentaren hinzugefügt.

Um aus dem Eingabestring ein passendes $ThreeAddressCodeInstruction$-Objekt zu generieren wird der Eingabestring im Konstruktor der Klasse nach Leerzeichen aufgeteilt und wie folgt die richtige Operation ausgewählt:

Listing 1: Konstruktor der Klasse ThreeAddressCodeInstruction

```
public ThreeAddressCodeInstruction(String rawInput, int address){
  ...
  String[] pieces = rawInput.split("␣");
  switch (pieces.length) {
    case 2 -> ... //Unbedingter Sprung goto X
    case 3 -> ... //Kopierbefehl X = Y
    case 4 -> { //entweder unaere Operation oder ein bedingter Sprung
      switch(pieces[2]){
        case "-" -> ... //Unaere Operation X = - Y
        case "goto" -> {//ein bedingter Sprung
          switch(pieces[0]){
            case "if" -> ... //bedingter Sprung if Y goto X
            case "ifFalse" -> ... //bedingter Sprung ifFalse Y goto X
          }
        }
      }
    case 5 -> ... //Binaere Operation X = Y op Z
    case 6 -> { //bedingter Sprung
      switch(pieces[2]){
        case "<" -> ...//if Y < Z goto X
        case ">" -> ...//if Y > Z goto X
        case "<=" -> ...//if Y <= Z goto X
        case ">=" -> ...//if Y >= Z goto X
        case "==" -> ...//if Y == Z goto X
        case "!=" -> ...//if Y != Z goto X
      }
    }
    ...
}
```

An den durch "..."markierten Stellen die zu gehörigen Attribute initialisiert.

Ausserdem wurden folgende Helfermethoden, welche im späteren Verlauf dieser Arbeit benötigt werden implementiert:

1. Die Methode $canJump()$ gibt den Wahrheitswert $wahr$ zurück, wenn die Instruktion springen kann oder immer springt, ansonsten gibt sie $falsch$ zurück.
2. Die Methode $writesValue()$ gibt den Wahrheitswert $wahr$ zurück, wenn die Instruk-

tion den Wert der in $destination$ gespeichert ist überschreibt. Ansonsten gibt sie $falsch$ zurück.

3. $getUsedIdentifiers()$ gibt die Menge der Werte in $source$ und $modifier$ zurück, wenn diese Variablen referenzieren und keine Konstanten sind.

4. $nextPossibleInstructionAddresses()$ gibt die Menge der nächsten möglichen Instruktionsaddressen zurück. Dies ist in den meißten Fällen einfach die nächste Addresse, wenn die aktuelle Instruktion jedoch ein unbedingter Sprung ist, ist es der Wert in $destination$, wenn die Instruktion ein bedingter Sprung ist, wird sowohl die nächste Addresse als auch der Wert in $destination$ zurückgegeben.

5. Die Methode $getRepresentationAsStringArray()$ gibt eine Repräsenationen der Instruktion als $String[]$ in der Form zurück, dass sie nacheinander die eingelesene Instruktion bilden zum Beispiel:

   Listing 2: "Rückgabewert von getRepresentationAsArray(true)"

   ```
   new String[]{address, destination, "=", source, op.getRepresentation()
   ```

   Die Methode $toString()$

2.4.2 Die ThreeAddressCode Klasse

2.4.3 Die BasicBlocks Klasse

2.5 Implementierung von Algorithmen

3 Implementierte Algorithmen

3.1 Erstellung von Grundblöcken aus 3-Address-Code

3.2 Erstellung eines Kontrollflussgraphen aus 3-Address-Code

3.3 Analyse von erreichenden Definitionen

3.4 Analyse von lebendigen Variablen bezüglich Grundblöcken

3.5 Analyse von lebendigen Variablen bezüglich einzelnen Instruktionen

4 Related Work

5 Evaluation

6 Future Work

7 Fazit

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Aho08] AHO, Alfred V. ; LEUSCHEL, Michael (Hrsg.): *Compiler Prinzipien, Techniken und Werkzeuge*. 2., aktualisierte Aufl., German language ed. München u.a. : Pearson Studium, 2008 (It, Informatik). https://digitale-objekte.hbz-nrw.de/storage/2008/03/03/file_131/2343468.pdf