Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

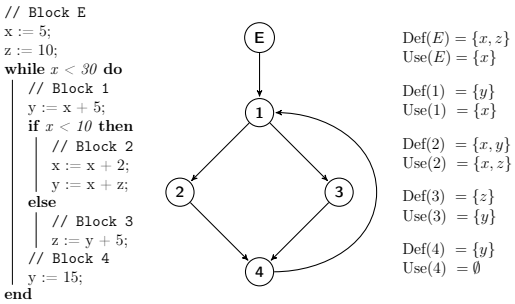
Lehrstuhl Prof. Dr.-Ing. Snelting

Verifizierte Berechnung von Datenabhängigkeiten

Bachelorarbeit von

Haoqian Zheng

an der Fakultät für Informatik



Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	Dipl.-Inform. Denis Lohner

Bearbeitungszeit: 18. Juni 2013 – 17. November 2013

English Abstract

This thesis introduces and verifies an executable, functional calculation of data dependencies on control flow graphs (CFG) using the theorem prover Isabelle. The algorithm of the calculation is based on a forward dataflow analysis and determines reaching definitions using the CFG of the program. Starting at the entry point of the CFG, the algorithm traverses the CFG recursively, while updating a state, that stores for each node its visible definitions, and looking for calculated data dependencies using the Def and Use sets of the CFG. The algorithm stores the result of the calculation inside a new graph containing the same nodes as the CFG and corresponding edges depending on whether a data dependency exists between two nodes or not. A main goal for the algorithm is its executability, so it is possible to export the calculation to some functional programming languages, e.g. Haskell, through the code generator of Isabelle. I also used the exported code to calculate data dependencies on a While-language.

I define the formalization of the algorithm inside a locale, that provides appropriate assumptions and requirements, e.g. for defining, traversing and building graphs, what I take from the bachelor thesis of Simon Kohlmeyer [8]. To accomplish the recursion of the algorithm the formalization uses the `while` function from the `While_Combinator` theory of the Isabelle HOL library.

This thesis also proves the correctness and completeness of the calculation, whereby I relate to the formal definition of data dependency from the dissertation of Daniel Wasserrab [13]. Since the algorithm uses the `while` function, the most important lemma for the proof is the `while_rule_lemma`, which is part of the `While_Combinator` theory. To apply this lemma, I introduce an appropriate invariant for the recursion and a well-founded relation based on the number of definitions visible in the state.

Abstract

In dieser Arbeit wurde eine Berechnung von Datenabhängigkeiten in Steuerflussgraphen mit dem Theorembeweiser Isabelle verifiziert. Der Algorithmus richtet sich dabei nach der formalen Definition von Datenabhängigkeit aus der Dissertation von Daniel Wasserrab [13] und ist so konzipiert, dass er in dem dort vorgestellten Slicing Framework einsetzbar ist. Die Berechnung basiert auf einer üblichen Datenflussanalyse, wie sie zum Beispiel in [10] zu finden ist. Für die dafür benötigten Graphen wurde das in der Bachelorarbeit von Simon Kohlmeyer [8] eingeführte Graphframework für Isabelle benutzt. Bei der Formalisierung des Algorithmus wurde außerdem auf seine Ausführbarkeit geachtet, weshalb es möglich ist, ihn über den Isabelle Codegenerator, in einige funktionale Sprachen, wie Haskell, zu exportieren. Als Anwendungsbeispiel wird der Algorithmus auf Steuerflussgraphen einer While-Sprache angewendet.

Inhaltsverzeichnis

1 Einleitung

Da Software in immer mehr sicherheitskritischen Anwendungen genutzt wird, ist vor allem die korrekte Funktionsweise der Programme wichtig. Die Methode des *Slicing* liefert eine Möglichkeit, Sicherheitsanalysen, wie das sogenannte *information flow control*, anzuwenden, um zu garantieren, dass vertrauliche Daten innerhalb des Programms nicht öffentlich und kritische Berechnungen nicht von außen manipuliert werden können [5]. Durch Slicing kann bestimmt werden, welche Stellen eines Programms die Ausführung einer bestimmten Anweisung beeinflussen können [14]. Dafür wird der Programmabhängigkeitsgraph (PDG) sowie der Systemabhängigkeitsgraph (SDG) des Programms verwendet. Der PDG enthält die Kontroll- und Datenabhängigkeiten des Programms als Kanten [3], während der SDG bestehende PDGs um sogenannte Summary Kanten ergänzt, die Auswirkungen von interprozeduralen Aufrufen darstellen.

Im Rahmen des Projekts *Quis custodiet* am Karlsruher Institut für Technologie soll ein verifizierter und ausführbarer Slicer entwickelt werden. In der Dissertation von Daniel Wasserrab [13] wurde die Korrektheit des Slicers bereits formal bewiesen. Außerdem hat Stefan Altmayer in seiner Bachelorarbeit [1] einen verifizierten und ausführbaren Algorithmus zur Berechnung von Summary Kanten erstellt. Ebenso existiert solch ein Algorithmus zur Berechnung von Kontrollabhängigkeiten aus der Bachelorarbeit von Maximilian Wagner [12]. Neben dem Slicer selbst, fehlt somit noch eine Möglichkeit zur Berechnung von Datenabhängigkeiten.

Diese Arbeit liefert eine Implementierung eines funktionalen Algorithmus zur Berechnung von Datenabhängigkeiten, die mit dem Theorembeweiser Isabelle verifiziert wurde und zum Aufbau eines PDG dienen kann. Die Verifizierung des Algorithmus bildet dabei den Hauptanteil der Arbeit, da für den Beweis in Isabelle formal korrektes Arbeiten erforderlich ist. Eine weitere Herausforderung bestand darin, die Ausführbarkeit des Algorithmus sicherzustellen. Für die Formalisierung musste daher auf die Verwendung einiger bestehender Konstrukte und Funktionen verzichtet werden, aus denen sich kein ausführbarer Code generieren lässt.

In dieser Arbeit wird außerdem dargestellt, warum der Algorithmus korrekt funktioniert, indem die zentralen Theoreme und Lemmata für den Beweis der Korrektheit erläutert werden. Dazu werden in Kapitel 2 zunächst die für das Verständnis des Algorithmus relevanten Grundlagen und Notationen eingeführt, wonach in Kapitel 3 die Berechnung sowie die Umsetzung in Isabelle vorgestellt wird. Die Vorgehensweise zum Korrektheitsbeweis des Algorithmus wird in Kapitel 4 dargestellt und Kapitel 5 geht schliesslich auf die konkrete Anwendung und Ausführung des Algorithmus ein.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Notationen eingeführt, die im weiteren Verlauf der Arbeit benutzt werden. Zunächst werden die Definitionen von Steuerflussgraphen und Datenabhängigkeiten erläutert und anschliessend die Isabelle Syntax vorgestellt.

2.1 Steuerflussgraphen

Ein Steuerflussgraph (CFG) dient zur Darstellung des Steuerflusses von Programmen. Dabei bilden Anweisungssequenzen ohne Steuerflussänderung die Knoten des CFG. Diese Sequenzen werden Basisblöcke genannt und können keine Sprunganweisungen enthalten. Ein Basisblock schliesst somit stets mit einer Sprunganweisung und eine Sprunganweisung hat stets den Beginn eines anderen Basisblocks als Ziel. Ist es möglich, dass Block B_j unmittelbar nach einem anderem Block B_i ausgeführt wird, so enthält der CFG eine gerichtete Kante von B_i nach B_j. Ein CFG hat einen definierten Startknoten und kann im Allgemeinen durch verschiedene Kontrollstrukturen, wie zum Beispiel Schleifen oder Sprunganweisungen, beliebige Zyklen enthalten. Für die Datenflussanalyse ist vor allem die Struktur des CFG von Bedeutung.

2.2 Datenabhängigkeiten

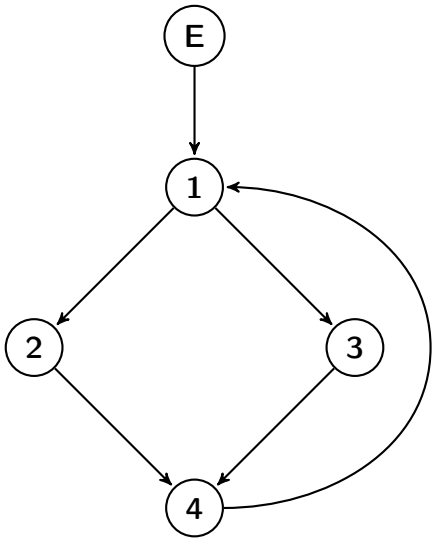
Für die Definition der Datenabhängigkeiten wird zunächst eine Erweiterung des CFG benötigt. Jeder Knoten erhält eine *Def*- und eine *Use*-Menge. Dabei enthält die *Def*-Menge sämtliche Variablen, die in diesem Knoten geschrieben, das heisst definiert beziehungsweise neu zugewiesen werden, und die *Use*-Menge alle Variablen, die in diesem Knoten gelesen, das heisst verwendet werden. Es besteht nun genau dann eine Datenabhängigkeit von Knoten m zu Knoten n, wenn in n eine Variable V geschrieben wird, die in m gelesen wird und es einen Pfad von n nach m gibt, auf dem kein Knoten V neu definiert. Die Definition aus n wird in diesem Fall als in m sichtbar bezeichnet, beziehungsweise die Definition erreicht m. Falls jedoch ein Knoten n' auf dem Pfad von n nach m existiert, in dem V neu definiert wird, so redefiniert n' die Definition aus Knoten n. Zu beachten ist, dass zwei unterschiedliche Knoten die gleiche Variable definieren können, dies jedoch zwei verschiedene Definitionen sind, da sich der definierende Knoten unterscheidet.

Die formale Definition von Datenabhängigkeit wurde der CFG Theorie von Wasserrab [13] entnommen. Dort bezeichnet `n influences V in n‘` eine Datenabhängigkeit von Knoten `n‘` zum Knoten `n` über die Variable `V`. Für eine Liste `as`, die Kanten des CFG enthält, wird mit `n -as→* n‘` notiert, dass über die Kanten in `as` ein Pfad von `n` nach `n’` existiert. Außerdem liefert `sourcenodes as` die Liste aller Startknoten der Kanten aus `as`. Die formale Definition für Datenabhängigkeit lautet

```
n influences V in n' = (∃a' as'. (V ∈ Def n) ∧ (V ∈ Use n') ∧
      (n -a'#as'→* n') ∧ (∀n'' ∈ set (sourcenodes as'). V ∉ Def n''))
```

Abbildung 2.1 zeigt einen CFG für ein einfaches Beispielprogramm.

```
// Beginn E
x := 5;
z := 10;
while x < 30 do
|  // Beginn 1
|  y := x + 5;
|  if x < 10 then
|  |  // Beginn 2
|  |  x := x + 2;
|  |  y := x + z;
|  else
|  |  // Beginn 3
|  |  z := y + 5;
|  // Beginn 4
|  y := 15;
end
```



$\text{Def}(E) = \{x, z\}$
$\text{Use}(E) = \{x\}$

$\text{Def}(1) = \{y\}$
$\text{Use}(1) = \{x\}$

$\text{Def}(2) = \{x, y\}$
$\text{Use}(2) = \{x, z\}$

$\text{Def}(3) = \{z\}$
$\text{Use}(3) = \{y\}$

$\text{Def}(4) = \{y\}$
$\text{Use}(4) = \emptyset$

Abbildung 2.1: Beispiel eines CFG mit *Def* und *Use* Mengen

In diesem Beispiel existieren folgende Datenabhängigkeiten, wobei die Notation (n, V, m) bedeutet, dass eine Datenabhängigkeit von Knoten n nach m über die Variable V existiert.

$(1, x, E),\ (2, x, E),\ (2, z, E),\ (3, y, 1),\ (1, x, 2),\ (2, z, 3),\ (2, x, 2)$

Die Datenabhängigkeit $(1, x, 2)$ existiert, da die in 2 erzeugte Definition von x über den Pfad $2 \to 4 \to 1$ nach 1 gelangt. Zu beachten ist außerdem, dass keine Datenabhängigkeit von Knoten 3 zu Knoten 4 existiert, da die Variable y auf jedem Pfad von 4 nach 3 in Knoten 1 redefiniert wird.

2.3 Datenflussanalyse

Um zu ermitteln, wie Informationen innerhalb des Programms weitergegeben werden und welche Abhängigkeiten daraus entstehen, wird eine Datenflussanalyse durchgeführt. Dabei spielt der CFG des Programms eine besondere Rolle. Dieser wird in der Datenflussanalyse, bei dem definierten Startknoten beginnend, traversiert, um zu untersuchen, wie sich die vom Programm erzeugten Daten zwischen den Blöcken verändern oder weitergegeben werden. Es wird hierbei zwischen der Vorwärtsanalyse und der Rückwärtsanalyse unterschieden, bei der der CFG in umgekehrter Reihenfolge durchlaufen wird und die in dieser Arbeit nicht näher betrachtet wird. Während der Traversierung des CFG wird der aktuelle (Wissens-)Zustand festgehalten. Dieser beinhaltet die Information, welche Daten aktuell an welchem Knoten vorliegen, und wird bei Weitergabe der Daten entsprechend aktualisiert. Dazu werden für jeden Block vier Mengen definiert. Mit $in(B)$ und $out(B)$ wird die Menge der Informationen bezeichnet, die am Anfang beziehungsweise Ende des Blocks B gültig sind. Die Menge der Informationen, die im Block B erzeugt beziehungsweise vernichtet werden, wird mit $gen(B)$ beziehungsweise $kill(B)$ notiert. Der Effekt eines Blocks B auf die Informationen aus $in(B)$ wird durch eine Transferfunktion f_B beschrieben. Es gilt also

$$out(B) = f_B(in(B)).$$

Mit Hilfe der Transferfunktionen lassen sich für jeden Block B die Mengen $in(B)$ und $out(B)$ bestimmen, womit weitere Aussagen zu dem Programm getroffen werden können. Dazu wird die Transferfunktion jedes Blocks, während der Traversierung des CFG, so lange angewendet, bis sich $in(B)$ und $out(B)$ für keinen Block B mehr verändert. Algorithmus 2.1 beschreibt diesen Vorgang.

```
foreach Block B do
    initialisiere in(B) und out(B);
end
repeat
    foreach Block B do
        in'(B) := in(B);  out'(B) := out(B);
        aktualisiere in(B);
        out(B) := f_B(in(B));
    end
until ∀B. in'(B) = in(B) und out'(B) = out(B);
```

Algorithmus 2.1: Grundprinzip der Datenflussanalyse

Eine ausführlichere Beschreibung von Datenflussanalysen findet sich zum Beispiel in [4] oder [10].

2.4 Isabelle Syntax

Um die Korrektheit der Berechnung zu verifizieren, wurde der Theorembeweiser Isabelle mit der Objektlogik *Higher Order Logic* (HOL) verwendet. HOL beinhaltet neben den üblichen Datentypen, wie natürliche Zahlen `nat` oder Wahrheitswerte `bool`, ein breites Maß an komplexeren, vordefinierten Datentypen, wie zum Beispiel Listen `'a list`, Mengen `'a set` oder Maps `'a` $\rightharpoonup$ `'b`. Dabei sind `'a` und `'b` Typvariablen.

Erhält man aus den Annahmen $A_1, \ldots, A_n$ die Konklusion E, so notieren wir dies mit dem Frege'schen Schlussstrich

$$\frac{A_1 \qquad \ldots \qquad A_n}{E}$$

oder alternativ mit der in Isabelle benutzten Schreibweise $[\![A_1; \ldots; A_n]\!] \Longrightarrow E$. Gilt eine Aussage `P` für alle Elemente `x`, so wird dies mit $\bigwedge$`x. P x` notiert. Fallunterscheidungen über einen Wert können entweder mit `if-then-else` oder über `case Wert of Alternative1` $\Rightarrow$ `Fall1 | Alternative2` $\Rightarrow$ `Fall2` ... umgesetzt werden.

Die leere Liste wird mit `[]` notiert und mit dem Operator `#` lässt sich über `x # xs` ein Element `x` vorne an die Liste `xs` anhängen. Außerdem lassen sich zwei Listen des gleichen Typs mit `@` aneinanderhängen. Das erste Element einer Liste `xs` lässt sich über `hd xs` ausgeben, während `tl xs` die Liste `xs` ohne das erste Element bezeichnet. Um die Menge aller Elemente in einer Liste `xs` zu erhalten, kann `set xs` benutzt werden.

Hat ein Element `a` den Typ `'a`, so wird dies mit `a :: 'a` notiert. Für die Typen `'a` und `'b` bezeichnet `'a` $\Rightarrow$ `'b` den Typ der Funktionen von `'a` nach `'b` und `'a` $\times$ `'b` den Produkttyp. Für diesen kann über die Projektionen `fst :: 'a` $\times$ `'b` $\Rightarrow$ `'a` und `snd :: 'a` $\times$ `'b` $\Rightarrow$ `'b` auf die einzelnen Elemente zugegriffen werden. Neue Datentypen lassen sich mit Hilfe des Schlüsselwortes `datatype`, vergleichbar mit dem Schlüsselwort `data` in Haskell, definieren und ebenso können Kurzschreibweisen für komplexere Datentypen über das Schlüsselwort `type_synonym` eingeführt werden.

Eine große Rolle in dieser Arbeit spielen Maps `'a⇀'b`. Sie ordnen Elemente des Typs `'a` Elementen des Typs `'b` zu, wobei es sich hierbei um partielle Funktionen handelt. In Isabelle wird dafür der Typ `'b option` verwendet, der die Konstruktoren `None` und `Some b` enthält. Ist ein Element `a` des Typs `'a` Schlüssel für ein Element `b ::'b` (das heisst dem Element `a` wird in der Map das Element `b` zugeordnet), so wird es auf `Some b ::'b option` abgebildet, ansonsten auf `None`. Für `Some b` liefert `the (Some b)` das Element `b` selbst. Da der Codegenerator von Isabelle jedoch keine zusammengesetzten Typen exportieren kann, wird in dieser Arbeit der Typ `('a, 'b) mapping` benutzt. Ein solches Mapping ist eine Kapselung der Map, was in Isabelle über ein *Lifting* realisiert wird [6]. Aus einem gegebenen Mapping `m` lässt sich über `Mapping.lookup m k` der dem Schlüssel `k` zugeordnete Wert (als `option` Typ) ausgeben und über `Mapping.update k x m` wird dem Schlüssel `k` in der Map `m` der neue Wert `x` zugeordnet. `Mapping.keys m` liefert die Menge aller Schlüssel der Map `m`, das heisst alle Werte `k`, für die `Mapping.lookup m k` nicht `None` zurückgibt. Das leere Mapping wird mit `Mapping.empty` bezeichnet. Eine ausführlichere Einleitung zu Isabelle findet sich zum Beispiel unter [11].

Für den Beweis wurde das Konzept der *Locales* [2] benutzt, die den Umgang mit parametrisierten Theorien in Isabelle ermöglichen. In einer Locale können über die Schlüsselwörter `fixes` und `assumes` lokale Definitionen und Annahmen getroffen werden, die im Nachhinein mit verschiedenen Parametern instanziiert werden können. Bestehende Locales lassen sich außerdem über den Operator `+` um weitere Annahmen und Definitionen erweitern.

2.5 Verwendete Frameworks

Für die Umsetzung eines CFG wurde das Graphframework von Kohlmeyer [8] verwendet. Es führt Graphen über Locales ein und definiert diverse Funktionen, um sie aufzubauen und zu traversieren. Relevant waren hierbei die Locales `graph_emtpy`, `graph_addEdge` sowie `graph_outEdges`.

```
locale graph =
fixes
  αe :: "'g ⇒ ('node × 'edgeD × 'node) set" and
  invar :: "'g ⇒ bool"

locale graph_empty = graph αe invar +
fixes empty :: "'g"
assumes
  empty_invar : "invar empty" and
```

```
  empty_correct : "αe empty = {}"

locale graph_addEdge = graph αe invar +
fixes addEdge :: "'g ⇒ 'node ⇒ 'edgeD ⇒ 'node ⇒ 'g"
assumes
  addEdge_invar : "invar g ⟹ invar (addEdge g f d t)" and
  addEdge_correct :
  "invar g ⟹ e ∈ αe (addEdge g f d t) ⟷ e = (f, d, t) ∨ e ∈ αe g"

locale graph_outEdges = graph αe invar +
fixes outEdges :: "'g ⇒ 'node ⇒ ('node × 'edgeD × 'node) list"
assumes
  outEdges_correct :
  "invar g ⟹ set (outEdges g n) = {(f, _, _). f = n} ∩ αe g"
```

Die Locales `graph_empty`, `graph_addEdge` und `graph_outEdges` erweitern `graph`, das die Funktionen `αe` und `invar` einführt. Dabei gibt `αe g` die Menge der Kanten (Tupel aus Startknoten, Kantenbezeichnung und Endknoten) des Graphen `g` zurück, während `invar g` eine Invariante des Graphen `g` bezeichnet, die erfüllt sein muss, damit `g` ein valider Graph ist. Dies könnte zum Beispiel die strukturelle Wohlgeformtheit der, dem Graphen zugrunde liegenden, Datenstruktur sein.

In `graph_empty` wird der leere Graph `empty` eingeführt und gefordert, dass er die Invariante erfüllt und eine leere Kantenmenge besitzt. Um einem Graphen Kanten hinzuzufügen, wird `graph_addEdge` benutzt, das die Funktion `addEdge` definiert und seine Korrektheit annimmt, während die Funktion `outEdges` aus `graph_outEdges` eine Liste von Kanten ausgibt, die von einem Knoten ausgehen.

Zusätzlich liefert das Graphframework konkrete Funktionen zur Instanziierung der Locales, die auf Rot-Schwarz-Bäumen basieren und mit deren Hilfe auch eine Interpretation des Algorithmus zur Berechnung der Datenabhängigkeiten erstellt wurde.

3 Berechnung der Datenabhängigkeiten

Dieses Kapitel erklärt den Algorithmus zur Berechnung der Datenabhängigkeiten und stellt seine Formalisierung in Isabelle vor.

Der Algorithmus basiert auf einer einfachen Vorwärtsanalyse. Für die Berechnung von Datenabhängigkeiten ist relevant, welche Definitionen einen Knoten erreichen. Daher ist für einen Block B vor allem die Menge $in(B)$ wichtig, die im Zustand abgespeichert wird. Zu Beginn sind in keinem Block sichtbare Definitionen vorhanden, sodass die *in*- und *out*-Mengen der Blöcke mit der leeren Menge initialisiert werden. Die im Block B generierten Definitionen sind genau die Variablen aus der entsprechenden *Def*-Menge, während alle eingehenden Definitionen dieser Variablen in der *kill*-Menge liegen, da sie von B redefiniert werden. Jeder Block ersetzt die Definitionen von Variablen aus seiner *Def*-Menge durch seine eigene Definition der Variable. Das heisst, die Transferfunktion für einen Block B ist gegeben durch

$$f_B(X) = gen(B) \cup (X \setminus kill(B)).$$

Die eingehenden Definitionen eines Blocks B sind genau die ausgehenden Definitionen seiner Vorgängerknoten $pred(B)$, es gilt also

$$in(B) = \bigcup_{B' \in pred(B)} out(B').$$

Ist die Traversierung des CFG abgeschlossen und der Zustand vollständig (das heisst für jeden Block B ist $in(B)$ bekannt), so lassen sich die Datenabhängigkeiten, durch einen Abgleich mit der *Use*-Menge des Knotens, ablesen. Diese werden als Kanten in einem neuen Graph mit den gleichen Knoten des CFG abgespeichert.

Um die Traversierung des CFG zu realisieren, nimmt eine Worklist die abzuarbeitenden Knoten auf und liefert den nächsten, für die Analyse relevanten, Knoten. Die eigentliche Traversierung des CFG wird durch Rekursion umgesetzt. Algorithmus 3.1 beschreibt die Transferfunktion für den Knoten m mit dem Vorgängerknoten n, wobei (V, n') die Definition der Variable V im Knoten n' bezeichnet.

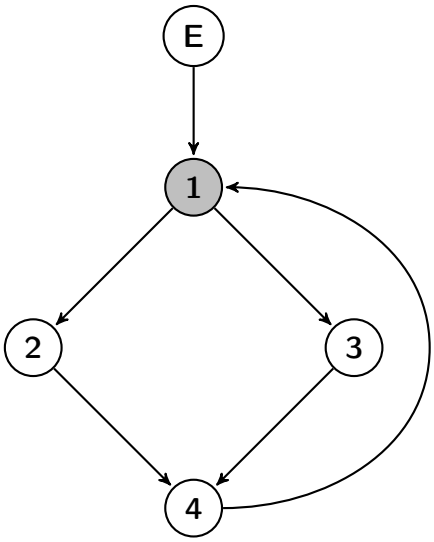
Da Definitionen, die einen Knoten einmal erreicht haben, nicht wieder gelöscht werden können, wird im Rekursionsschritt ebenfalls überprüft, ob durch die Aktualisierung des Zustands eine neue Datenabhängigkeit berechnet wurde. Ist dies der Fall, so wird diese direkt dem Ergebnisgraph hinzugefügt. Am Ende des Rekursionsschrittes wird der aktuell betrachtete Knoten aus der Worklist entfernt.

foreach $V \in \mathit{Def}(n)$ **do**
  Füge im Zustand für m die Definition (V, n) hinzu
end
foreach (V, x) *sichtbar in* n **do**
  if $V \notin \mathit{Def}(n)$ **then**
    `// (V, x) ist nicht in der kill-Menge von n`
    Füge im Zustand für m die Definition (V, x) hinzu
  else
    `// Der Zustand wird nicht verändert, da (V, x) in der kill-`
    `// Menge von n liegt.`
  end
end

Algorithmus 3.1: Aktualisierung des Zustands

Falls der Zustand verändert wurde, so werden außerdem seine Nachfolgeknoten an die Worklist gehängt. Da sich der Zustand nach endlich vielen Schritten nicht mehr verändert, nämlich dann, wenn er alle sichtbaren Definitionen enthält, wird die Worklist nach endlich vielen Schritten leer sein. Der Algorithmus terminiert in diesem Fall.

In den Abbildungen 3.1, 3.2 und 3.3 wird jeweils das Ergebnis eines Rekursionsschrittes dargestellt. Der aktuell bearbeitete Knoten ist grau markiert.



$\mathrm{Def}(E) = \{x, z\}$
$\mathrm{Use}(E) = \{x\}$

$\mathrm{Def}(1) = \{y\}$
$\mathrm{Use}(1) = \{x\}$

$\mathrm{Def}(2) = \{x, y\}$
$\mathrm{Use}(2) = \{x, z\}$

$\mathrm{Def}(3) = \{z\}$
$\mathrm{Use}(3) = \{y\}$

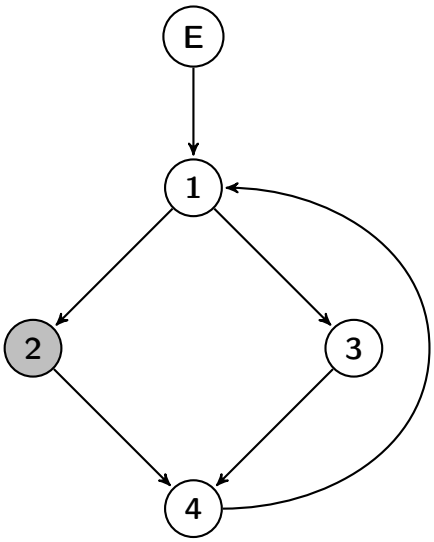
$\mathrm{Def}(4) = \{y\}$
$\mathrm{Use}(4) = \emptyset$

aktueller Knoten
1

Worklist
[2, 3]

Zustand
E: {}
1: {(x, E), (z, E)}
2: {(x, E), (z, E), (y, 1)}
3: {(x, E), (z, E), (y, 1)}
4: {}

Abbildung 3.1: Zustand nach Bearbeitung von Knoten 1



$\text{Def}(E) = \{x, z\}$
$\text{Use}(E) = \{x\}$

$\text{Def}(1) = \{y\}$
$\text{Use}(1) = \{x\}$

$\text{Def}(2) = \{x, y\}$
$\text{Use}(2) = \{x, z\}$

$\text{Def}(3) = \{z\}$
$\text{Use}(3) = \{y\}$

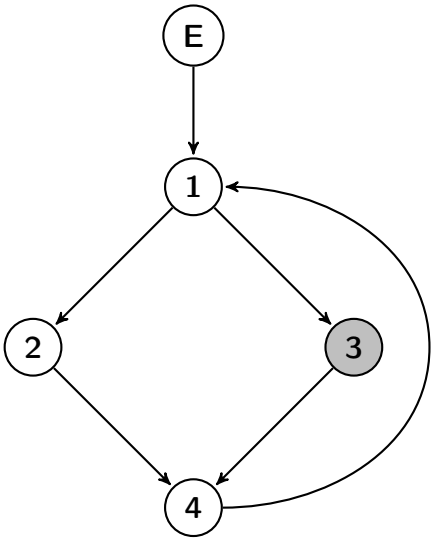
$\text{Def}(4) = \{y\}$
$\text{Use}(4) = \emptyset$

aktueller Knoten
2

Worklist
[3, 4]

Zustand
E: {}
1: {(x, E), (z, E)}
2: {(x, E), (z, E), (y, 1)}
3: {(x, E), (z, E), (y, 1)}
4: {(z, E), (x, 2), (y, 2)}

Abbildung 3.2: Zustand nach Bearbeitung von Knoten 2



$\text{Def}(E) = \{x, z\}$
$\text{Use}(E) = \{x\}$

$\text{Def}(1) = \{y\}$
$\text{Use}(1) = \{x\}$

$\text{Def}(2) = \{x, y\}$
$\text{Use}(2) = \{x, z\}$

$\text{Def}(3) = \{z\}$
$\text{Use}(3) = \{y\}$

$\text{Def}(4) = \{y\}$
$\text{Use}(4) = \emptyset$

aktueller Knoten
3

Worklist
[4, 4]

Zustand
E: {}
1: {(x, E), (z, E)}
2: {(x, E), (z, E), (y, 1)}
3: {(x, E), (z, E), (y, 1)}
4: {(z, E), (x, 2), (y, 2),
(z, 3)}

Abbildung 3.3: Zustand nach Bearbeitung von Knoten 3

3.1 Formalisierung der Berechnung in Isabelle

Die Berechnung ist innerhalb einer Erweiterung der in Kapitel 2.5 eingeführten `graph` Locales von Kohlmeyer [8] definiert. Zusätzlich wird die Endlichkeit der Knoten im Graph gefordert. Um einen neuen Graph zu erzeugen und in ihn die Datenabhängigkeiten einzutragen, werden `graph_empty` und `graph_addEdge` mit anderen Parametern als `graph_outEdges` instanziiert, das auf dem übergebenen CFG arbeitet. Dadurch ist es möglich, einen vom CFG unabhängigen Ergebnisgraphen zurückzugeben.

```
locale DataDependencyGraph
  = graph_empty αe' invar' empty' +
  graph_addEdge αe' invar' addEdge' +
  graph_outEdges αe invar outEdges +
  fixes
    entry :: "'graph ⇒ 'node"
  assumes nodes_finite: "finite (fst ` αe g ∪ (snd ∘ snd) ` αe g)"
```

Die Darstellung von Variablen erfolgt durch Zeichenketten aus Isabelle.

```
type_synonym variable = String.literal
```

Die *Def*- und *Use*-Mengen werden mit Hilfe von Mappings realisiert. Dabei wird einem Knoten `'node` eine Liste an Variablen `variable list` zugeordnet, die die Elemente aus der *Def*- beziehungsweise *Use*-Menge enthalten.

```
type_synonym 'node dd_def = "('node, variable list) mapping"
type_synonym 'node dd_use = "('node, variable list) mapping"
```

Ebenso werden sichtbare Definitionen über ein Mapping umgesetzt. Die definierte Variable bildet den Schlüssel, dem der definierende Knoten zugeordnet wird. Da es jedoch möglich ist, dass mehrere Knoten die gleiche Variable definieren, wird einer Variable eine Liste von Knoten zugewiesen.

```
type_synonym 'node dd_visibleDefs = "(variable, 'node list) mapping"
```

Der Zustand für den Algorithmus speichert für jeden Knoten die aktuell sichtbaren Definitionen ab, weshalb dieser mit einem Mapping von Knoten nach sichtbaren Definitionen realisiert wird.

```
type_synonym 'node dd_state = "('node, 'node dd_visibleDefs) mapping"
```

Für die Umsetzung der Rekursion wurde die Funktion

```
while :: "('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a"
```

aus der `While_Combinator` Theory benutzt, die Teil der HOL-Bibliothek von Isabelle ist. Diese Funktion erfüllt die folgende Gleichung

```
while b c s = (if b s then while b c (c s) else s)
```

und wendet auf einen Rekursionszustand `s :: 'a` so lange die Funktion `c` an, bis die Abbruchbedingung `b s` erfüllt ist. Für den Algorithmus wird daher ein Rekursionszustand benötigt, der alle Informationen enthält, die sich im Laufe der Rekursion verändern. Dies ist die Worklist, der aktuelle Zustand sowie der Ergebnisgraph. Da die Zusammensetzung der neuen Worklist von der Veränderung des Zustands abhängt, wird zum Abgleich außerdem noch der Zustand aus dem vorherigen Schritt benötigt. Es ergibt sich somit für den Rekursionszutand

```
type_synonym state =
  "('node list × 'node dd_state × 'node dd_state × 'dd_graph)".
```

In jedem Rekursionsschritt muss überprüft werden, ob sich der Zustand verändert hat. Für diesen Fall werden die Nachfolgeknoten des aktuellen Knotens an die Worklist angehängt. Diese Aufgabe erfüllt die Funktion

```
manageQueue :: "'node list ⇒ 'node list ⇒ 'node dd_state
                ⇒ 'node dd_state ⇒ 'node list" where
  "manageQueue x [] l n = x"
| "manageQueue x y l n  = (case l = n of True ⇒ x | False ⇒ x @ y)".
```

Sie überprüft, ob sich die übergebenen Zustände `l` und `n` unterscheiden und fügt in dem Fall die zweite Liste an die Erste an.

Der wichtigste Teil im Rekursionsschritt besteht darin, den Zustand zu aktualisieren. Dafür kann es nötig sein, zwei `dd_visibleDefs` zusammenzufügen, nämlich dann, wenn in einem Knoten, der bereits Definitionen enthält, weitere Definitionen eingetragen werden (wie zum Beispiel in Knoten 4, während des Schrittes von Abbildung 3.2 nach Abbildung 3.3). Dafür wurde die Funktion

```
merge :: "('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a,'b) mapping ⇒ ('a,'b) mapping
          ⇒ ('a,'b) mapping"
```

definiert, die zwei Mappings zusammenfügt. Da es möglich ist, dass das erste Mapping ein anderes Element auf einen Schlüssel `k` abbildet, als das Zweite, wird eine Funktion übergeben, die in diesem Fall das Element für den Schlüssel `k` angibt.

Falls dies bei der Zusammenführung zweier `dd_visibleDefs` auftritt, so müssen die beiden Listen aneinander gehängt und eventuell aufgetretene Duplikate entfernt werden, was durch die Funktion `deduplicateList` realisiert wird.

Damit kann nun eine Funktion definiert werden, die für einen Knoten und seine *Def*-Menge den Zustand für jeden Knoten einer Liste aktualisiert. Die Hilfsfunktion

```
updateStateList :: "'node ⇒ dd_definition list
                    ⇒ 'node dd_visibleDefs ⇒ 'node dd_visibleDefs"
```

trägt dabei jede Variable aus dem `variable list` Parameter in die übergebene `'node dd_visibleDefs`, mit dem `'node` Parameter als definierenden Knoten, ein. Die Funktion, die den Zustand aktualisiert ergibt sich durch

```
updateState :: "'node ⇒ dd_definition list option ⇒ 'node list
                ⇒ 'node dd_state ⇒ 'node dd_state" where
  "updateState n d [] s = s"
| "updateState n None (x#xs) s = (case Mapping.lookup s x of
    None ⇒
      (case Mapping.lookup s n of
        None ⇒ updateState n None xs (Mapping.update x Mapping.empty s) |
        Some i ⇒ updateState n None xs (Mapping.update x i s)) |
    Some is ⇒
      (case Mapping.lookup s n of
        None ⇒ updateState n None xs (Mapping.update x is s) |
        Some i ⇒ updateState n None xs (Mapping.update x
         (Mapping_ext.merge (λ a b1 b2. deduplicateList(b1 @ b2)) i is) s))
    )"
| "updateState n (Some d) (x#xs) s = (case Mapping.lookup s x of
    None ⇒
      (case Mapping.lookup s n of
        None ⇒ updateState n (Some d) xs (Mapping.update x
            (updateStateList n d Mapping.empty) s) |
        Some i ⇒ updateState n (Some d) xs (Mapping.update x
            (updateStateList n d i) s)) |
    Some is ⇒
      (case Mapping.lookup s n of
        None ⇒ updateState n (Some d) xs (Mapping.update x
          (Mapping_ext.merge (λ a b1 b2. deduplicateList(b1 @ b2))
          (updateStateList n d Mapping.empty) is) s) |
        Some i ⇒ updateState n (Some d) xs (Mapping.update x
          (Mapping_ext.merge (λ a b1 b2. deduplicateList(b1 @ b2))
```

```
        (updateStateList n d i) is) s))
  )"
```

Sie arbeitet rekursiv die Liste der übergebenen Knoten `x#xs` ab. Dabei wird jeweils überprüft, ob für `n` und `x` bereits sichtbare Definitionen eingetragen sind, das heisst, ob `Mapping.lookup s n` $\neq$ `None` beziehungsweise `Mapping.lookup s x` $\neq$ `None` gilt und führt diese eventuell mit `merge` zusammen.

Um den Ergebnisgraphen aufzubauen, wurde die Funktion

```
addDdEdgesTo :: "'dd_graph ⇒ 'node ⇒ dd_edge_type
                 ⇒ 'node list ⇒ 'dd_graph"
```

definiert. Der Aufruf `addDdEdgesTo g n e ns` fügt dem Graphen `g` für jeden Knoten `n'` in `ns` jeweils eine Kante mit der Bezeichnung `e` von `n` nach `n'` zu. Mit den folgenden Funktionen `checkDdNode` und `checkDd` wird geprüft, ob eine neue Datenabhängigkeit ermittelt wurde, und diese zum Ergebnisgraphen hinzugefügt.

```
checkDdNode :: "'node ⇒ dd_definition list option
      ⇒ 'node dd_visibleDefs option ⇒ 'dd_graph ⇒ 'dd_graph" where
  "checkDdNode n u None g = g"
| "checkDdNode n None i g = g"
| "checkDdNode n (Some []) i g = g"
| "checkDdNode n (Some (u#us)) (Some i) g = (case (Mapping.lookup i u) of
     None ⇒ checkDdNode n (Some us) (Some i) g |
     Some n' ⇒ checkDdNode n (Some us) (Some i)
         (addDdEdgesTo g n (DataDependency u) n'))"

checkDd :: "'node list ⇒ 'node dd_use ⇒ 'node dd_state ⇒ 'dd_graph
             ⇒ 'dd_graph" where
  "checkDd [] u s g = g"
| "checkDd (n#ns) u s g = checkDd ns u s
      (checkDdNode n (Mapping.lookup u n) (Mapping.lookup s n) g)"
```

Die Funktion `checkDdNode` geht dafür rekursiv durch die *Use*-Menge des übergebenen Knotens und prüft, ob eine sichtbare Definition dieser Variablen in der übergebenen `dd_visibleDefs` vorhanden ist. In diesem Fall werden die entsprechenden Kanten zum Ergebnisgraph hinzugefügt. Die Funktion `checkDd` ruft für die übergebene Liste von Knoten rekursiv jeweils `checkDdNode` auf.

Der Aufruf `getOutNodes g n` liefert alle Nachfolgeknoten von `n` im Graphen `g`. Mit Hilfe dieser Funktionen wurde nun die Funktion

```
traverseOutEdges :: "'node dd_def ⇒ 'node dd_use ⇒ 'graph
                     ⇒ state ⇒ state" where
  "traverseOutEdges d u g (ns, lastState, curState, result) =
    (let newState = (updateState (hd ns) (Mapping.lookup d (hd ns))
      (getOutNodes g (hd ns)) curState)
    in ((manageQueue (tl ns) (getOutNodes g (hd ns)) curState newState),
      curState, newState,
      (checkDd (getOutNodes g (hd ns)) u newState result)))"
```

definiert, die einen Rekursionsschritt ausführt, indem die Funktionen `manageQueue`, `updateState` und `checkDd` auf die entsprechenden Teile des Rekursionszustands `state` aufgerufen werden.

Die folgende Funktion berechnet die Datenabhängigkeiten für einen Graph `'graph` und fügt diese als Kanten zum Ergebnisgraph `'dd_graph` hinzu.

```
addDataDependencies :: "'graph  ⇒ 'node dd_def ⇒ 'node dd_use
                        ⇒ 'dd_graph ⇒ 'dd_graph" where
  "addDataDependencies g d u result = (snd ∘ snd ∘ snd)
    (While_Combinator.while (λ s. (fst s ≠ [])) (traverseOutEdges d u g)
    ([entry g], Mapping.empty, (Mapping.update (entry g)
      Mapping.empty Mapping.empty), result))"
```

Die Abbruchbedingung der Rekursion ist erfüllt, wenn die Worklist leer ist. Zu Beginn der Rekursion enthält die Worklist nur den Startknoten des CFG, in dem am Anfang keine Definitionen sichtbar sind. Das heisst, dem Startknoten ist das leere Mapping zugordnet.

4 Beweise zum Algorithmus

Dieses Kapitel stellt die, in der Arbeit bewiesenen, Aussagen bezüglich der Berechnung vor und erläutert diese. Der Beweis der Hauptaussage teilt sich dabei in vier wesentliche Teile auf und verwendet eine Rekursionsinvariante, auf die in diesem Kapitel näher eingegangen wird.

Für den Beweis der Korrektheit des Algorithmus wurde die formale Definition für Datenabhängigkeiten aus der `CFG` Theory der Dissertation von Wasserrab [13] benutzt. Es wurde gezeigt, dass in einem CFG `g` genau dann eine Datenabhängigkeit von Knoten `n'` zu Knoten `n` über die Variable `V` existiert, wenn die entsprechende Kante im Ergebnisgraph, der von dem Algorithmus erstellt wird, vorhanden ist. Der Beweis wird in einer Locale vollzogen, die die formale Definition von Wasserrab aus der `CFG_wf` Locale mit der Formalisierung des Algorithmus vereinigt und ihre Konsistenz untereinander annimmt.

```
locale DataDependencyProof = DataDependencyGraph + CFG_wf +
assumes
  all_nodes_reachable : "n ∈ getNodes g ⟹ ∃as. entry g -as→* n" and
  entry_in_graph : "entry g ∈ getNodes g"
```

Es wurde zusätzlich angenommen, dass der definierte Startknoten im CFG liegt und für jeden Knoten im CFG ein Pfad vom Startknoten aus zu ihm existiert, das heisst der CFG enthält keinen nicht erreichbaren Code. Die Funktionen `defConsist :: ‘’graph ⇒ 'node dd_def ⇒ bool‘‘` und `useConsist :: ‘’graph ⇒ 'node dd_use ⇒ bool‘‘` überprüfen, ob die Definitionen der *Def*- und *Use*-Menge zwischen den beiden Locales konsistent sind. Der leere Graph wird mit `empty'` bezeichnet. Die Funktion

```
ddep_graph :: "'graph ⇒ dd_def ⇒ dd_use ⇒ 'dd_graph" where
  "ddep_graph g d u = addDataDependencies g d u empty'"
```

erstellt für den Graphen `g`, mit den *Def*- beziehungsweise *Use*-Mengen `d` beziehungsweise `u`, den vom Algorithmus berechneten Ergebnisgraphen, der die Datenabhängigkeiten von `g` als Kanten enthält. Die zu zeigende Hauptaussage lautet

```
theorem dataDependencyGraph_correct :
assumes "defConsist g d" "useConsist g u" "invar g"
  shows "(n influences V in n') = ((n', DataDependency V, n)
         ∈ αe' (ddep_graph g d u))".
```

Das wesentliche Lemma, das für den Beweis dieser Aussage benutzt wird, ist

```
lemma while_rule_lemma:
⟦P st; ⋀st. ⟦P st; b st⟧ ⟹ P (c st);
  ⋀st. ⟦P st; ¬ b st⟧ ⟹ Q st; wf r;
  ⋀st. ⟦P st; b st⟧ ⟹ (c st, st) ∈ r⟧ ⟹ Q (while b c st)
```

aus der `While_Combinator` Theorie. Um eine Aussage `Q` über `while b c st` zu zeigen, wird eine Rekursionsinvariante `P` benötigt. Diese muss für den Anfangszustand gelten. Ist die Abbruchbedingung nicht erfüllt, so muss gezeigt werden, dass die Invariante nach einem Rekursionsschritt immer noch gilt. Falls die Abbruchbedingung allerdings erfüllt ist, so muss die zu zeigende Aussage `Q` gelten. Zusätzlich muss durch die Benutzung einer wohlfundierten Relation `r` gezeigt werden, dass die Rekursion terminiert. Mit Hilfe dieser Relation wird gezeigt, dass sich eine bestimmte, nichtnegative Größe bei jedem Rekursionsaufruf verkleinert, sodass die Rekursion zwangsläufig nach endlich vielen Schritten terminieren muss. Dabei kann die Rekursionsinvariante als Annahme benutzt werden.

4.1 Die Rekursionsinvariante

Bei der Anwendung des `while_rule_lemma` erfüllt die Rekursionsinvariante zwei Aufgaben. Einerseits muss aus ihr die Hauptaussage herleitbar sein, wenn die Abbruchbedingung erfüllt ist, andererseits muss sie die nötigen Voraussetzungen für die Termination der Rekursion beinhalten. Allerdings muss es dabei noch möglich sein zu zeigen, dass die Invariante in jedem Rekursionsschritt erhalten bleibt. In diesem Teil soll nun die Rekursionsinvariante für `(ws, s', s, dd_graph) :: state` vorgestellt werden, das heisst die Worklist wird nun mit `ws`, der letzte beziehungsweise aktuelle Zustand mit `s'` beziehungsweise `s` und der Ergebnisgraph mit `dd_graph` bezeichnet.

Um die Hauptaussage zeigen zu können, muss die Rekursionsinvariante die nötigen Voraussetzungen enthalten, damit im Zustand eine Definition für einen Knoten sichtbar ist. In diesem Fall erfolgt ein Abgleich mit den entsprechenden *Def*- und *Use*-Mengen und eine vorhandene Datenabhängigkeit wird im Ergebnisgraph eingetragen. Die Invariante enthält also die in Abbildung 4.1 dargestellte Aussage. Das heisst, wenn im Zustand `s` für den Knoten `n'` eine Definition der Variable `V` aus `n` sichtbar ist und `V` in der *Def*-Menge von `n` und der *Use*-Menge von `n'` liegt, dann enthält der Ergebnisgraph `dd_graph` die entsprechende Kante.

Es wird nun eine Bedingung gesucht, damit eine Definition aus `n` im Zustand für den Knoten `n'` eingetragen ist. Damit dies überhaupt möglich sein kann, muss der Knoten `n` in einem vorherigen Rekursionsschritt besucht worden sein. Ist ein

```
∀n ∈ getNodes g. ∀V. ∀n' ∈ getNodes g.
  (
    (n', DataDependency V, n) ∈ αe' dd_graph
    ⟷
    (
     (∃vd ns. Mapping.lookup s n' = Some vd ∧
       Mapping.lookup vd V = Some ns ∧ n ∈ set ns)
     ∧ V ∈ Def n ∧ V ∈ Use n'
    )
  )
```

Abbildung 4.1: Äquivalenz zur Existenz einer Kante im Ergebnisgraph

Knoten `n` nicht in der Worklist und gibt es von keinem Knoten aus der Worklist einen Pfad zu `n`, so muss `n` bereits besucht worden sein, da die Rekursion nur noch Nachfolgeknoten von Knoten aus der Worklist besucht und nach Voraussetzung jeder Knoten im CFG erreichbar ist. Ein Knoten `n` wurde genau dann besucht, wenn `Mapping.lookup s n` $\neq$ `None` gilt, denn sobald ein Knoten besucht wurde, wird ihm über die Funktion `updateState` ein `dd_visibleDefs` Mapping zu geordnet. Somit ist die Aussage aus Abbildung 4.2 ein weiterer Teil der Invariante und liefert eine hinreichende Bedingung dafür, ob ein Knoten bereits besucht wurde.

```
∀n ∈ getNodes g.  (n ∉ set ws ∧ (∀n' ∈ set ws. ¬ (∃as. n' -as→* n)))
  ⟶ Mapping.lookup s n ≠ None)
```

Abbildung 4.2: hinreichende Bedingung, damit ein Knoten besucht wurde

Damit eine Definition von `V` in `n` im Zustand `s` für den Knoten `n'` eingetragen sein kann, muss es im CFG einen Pfad `p` von `n` nach `n'` geben, auf dem `V` nicht redefiniert wird. Zusätzlich muss der Knoten `n` in einem vorherigen Rekursionsschritt bereits besucht worden sein und ein Teilpfad `p'` von `p` existieren, der in einem Knoten `w'` beginnt und in `n'` endet. Die Definition muss in `w'` sichtbar sein und kein Knoten auf dem Pfad `p'` darf in der Worklist enthalten sein. In diesem Fall hat der Algorithmus den Teilpfad `p'` bereits komplett durchlaufen, da `w'` bereits besucht wurde, und Nachfolgeknoten stets zur Worklist hinzugefügt werden. Da der Teilpfad `p'` durchlaufen wurde und `V` auf `p'` nicht redefiniert wird, ist die Definition im Zustand auch für den Knoten `n'` eingetragen. Für die Formalisierung wurde diese Bedingung zur Einfachheit in zwei Spezialfälle aufgeteilt:

1. Kein Knoten auf dem Pfad `p` ist in der Worklist enthalten. Dies ist der Spezialfall, dass `p = p'` beziehungsweise `n = w'` gewählt werden kann.

2. Es gibt einen Knoten `w` auf dem Pfad `p`, der in der Worklist steht und einen

Nachfolgeknoten `w'` von `w`, in dem die Definition sichtbar ist. Außerdem ist kein Knoten auf dem Pfad von `w'` nach `n'` in der Worklist enthalten.

Abbildung 4.3 zeigt diese beiden Spezialfälle formal.

1. $\forall$ *w ∈ set ws. w ∉ set (sourcenodes (a#as))*
2. $\exists$ *w w' b bs.*
   w ∈ set ws ∧ w' ∈ set (getOutNodes g w) ∧
   (∃ vd ns. Mapping.lookup s w' = Some vd
   ∧ Mapping.lookup vd V = Some ns ∧ n ∈ set ns) ∧
   w' -b#bs→ n' ∧ ∀ w ∈ set ws. w ∉ set (sourcenodes (b#bs))*

Abbildung 4.3: Formalisierung der beiden Spezialfälle

Es ergibt sich somit für die Invariante zusätzlich die Aussage aus Abbildung 4.4.

```
∀n ∈ getNodes g. ∀V. ∀n' ∈ getNodes g. Mapping.lookup s n ≠ None ⟶
  (
    (∃vd ns. Mapping.lookup s n' = Some vd
      ∧ Mapping.lookup vd V = Some ns ∧ n ∈ set ns)
  ⟷
    ( V ∈ Def n ∧ ∃a as.
      (
        n -a#as→* n' ∧ ∀n'' ∈ set (sourcenodes as). V ∉ Def n''
        ∧
        (
          ∀w ∈ set ws. w ∉ set (sourcenodes (a#as))
          ∨
          ∃ w w' b bs.
            w ∈ set ws ∧ w' ∈ set (getOutNodes g w) ∧
            (∃ vd ns. Mapping.lookup s w' = Some vd
              ∧ Mapping.lookup vd V = Some ns ∧ n ∈ set ns) ∧
            w' -b#bs→* n' ∧ ∀w ∈ set ws. w ∉ set (sourcenodes (b#bs))
        )
      )
    )
  )
```

Abbildung 4.4: Äquivalenz zum Eintrag einer sichtbaren Definition im Zustand

Um zu garantieren, dass sämtliche Knoten, die vom Algorithmus verarbeitet werden, im CFG liegen enthält die Invariante außerdem die Aussagen `set ws` $\subseteq$ `getNodes g`, `Mapping.keys s` $\subseteq$ `getNodes g` und

```
∀ vd n V ns.(Mapping.lookup s n = Some vd ∧ Mapping.lookup vd V = Some ns)
  ⟶ set ns ⊆ getNodes g.
```

Für die Termination des Algorithmus wird die Endlichkeit der Schlüssel des Zustands `s`, sowie jeder sichtbaren Definition `dd_visibleDefs` in `s` benötigt.

```
finite (Mapping.keys s) ∧
∀ k ∈ Mapping.keys. finite (Mapping.keys (the (Mapping.lookup s k)))
```

Außerdem dürfen die `dd_visibleDefs` aus `s` keine Listen mit Duplikaten enthalten.

```
∀ ks d. Mapping.lookup s ks = Some d ⟶
  ∀ k ∈ Mapping.keys d. distinct (the (Mapping.lookup d k))
```

Dies würde ansonsten zur Folge haben, dass zwei Zustände, die die gleichen Einträge haben, formal als verschiedene Zustände aufgefasst werden. Für den Beweis der Termination des Algorithmus ist es notwendig, dies zu verhindern.

4.2 Korrektheit und Vollständigkeit des Algorithmus

Um die Hauptaussage, die die Korrektheit und Vollständigkeit des Algorithmus impliziert, zu zeigen, müssen drei Aussagen gezeigt werden.

1. Die Invariante ist für den Rekursionszustand zu Beginn der Rekursion erfüllt.
2. Die Invariante bleibt bei einem Rekursionsschritt erhalten.
3. Ist die Abbruchbedingung erfüllt, so folgt aus der Invariante die Hauptaussage.

Im Folgenden wird der Beweis dieser drei Teile erläutert.

4.2.1 Gültigkeit der Invariante zu Beginn der Rekursion

Der Rekursionszustand zu Beginn der Rekursion ist gegeben durch

```
([entry g], Mapping.empty,
 (Mapping.update (entry g) Mapping.empty Mapping.empty), empty')
```

das heisst, die Worklist enthält nur den Startknoten und im Zustand ist nur für den Startknoten ein leeres `dd_visibleDefs` eingetragen.

Die erste Aussage der Invariante aus Abbildung 4.1 ist erfüllt, da `empty'` keine Knoten enthält und für keinen Knoten `n'` sichtbare Definitionen eingetragen sind. Die Bedingung aus Abbildung 4.2 ist ebenfalls erfüllt, da nach Voraussetzung jeder Knoten von `entry g` aus erreichbar ist, das heisst $\forall$`n' ∈ set ws. ¬ (∃as. n' -as→* n)` ist für kein `n ∈ getNodes g` erfüllt, womit die Implikation und damit die Bedingung wahr ist. Ebenso ist die dritte Aussage aus Abbildung 4.4 erfüllt, denn nur für `n = entry g` gilt `Mapping.lookup s n ≠ None`. Der linke Teil der Äquivalenz

```
∃ vd ns. Mapping.lookup s n' = Some vd
      ∧ Mapping.lookup vd V = Some ns ∧ n ∈ set ns
```

ist immer falsch, ebenso wie die beiden Spezialfälle aus Abbildung 4.3. Der erste Spezialfall ist nicht erfüllt, da jeder Pfad `a#as` bei `n = entry g` beginnt, der jedoch in der Worklist enthalten ist. Auch der zweite Spezialfall tritt nicht auf, denn für jeden Nachfolgeknoten `w'` von `entry g` gilt `Mapping.lookup s w' = None`.

Die restlichen Aussagen der Invariante sind ebenso erfüllt, wie man leicht einsieht.

4.2.2 Erhaltung der Invariante

Um die Erhaltung der Invariante zu zeigen, müssen die Aussagen mit folgenden Substitutionen bewiesen werden, wobei `d :: dd_def` und `u :: dd_use` die übergebenen *Def*- beziehungsweise *Use*-Mengen bezeichnen.

```
ws       := manageQueue (tl ws) (getOutNodes g (hd ws)) s
              (updateState (hd ws) (Mapping.lookup d (hd ws))
                (getOutNodes g (hd ws)) s)

s'       := s

s        := updateState (hd ws) (Mapping.lookup d (hd ws))
              (getOutNodes g (hd ws)) s

dd_graph := checkDd (getOutNodes g (hd ns)) u
              (updateState (hd ws) (Mapping.lookup d (hd ws))
                (getOutNodes g (hd ws)) s) dd_graph
```

Dafür wurde in allen Beweisen eine Regelinduktion nach der Funktion `updateState`

durchgeführt.

Die erste Aussage aus Abbildung 4.1 folgt sofort aus der Definition der Funktion `checkDd`, da diese den übergebenen Zustand komplett durchläuft und genau dann eine Kante in den Ergebnisgraph einfügt, wenn die sichtbare Definition im Zustand eingetragen ist und die Variable in der *Use*-Menge steht.

Die Bedingung in Abbildung 4.2 lässt sich durch Betrachtung der neuen Worklist zeigen. Aus ihr wird der zuletzt bearbeitete Knoten `n` entfernt, das heisst dies ist der einzige Knoten, für den die Bedingung nicht bereits im letzten Rekursionsschritt gelten musste. Da `n` der zuletzt bearbeitete Knoten ist, wurde er besucht. Dementsprechend gilt `Mapping.lookup s n` $\neq$ `None` und somit die Implikation. Das Hinzufügen der Nachfolgeknoten von `n` ist für die Aussage nicht relevant, denn falls es keinen Pfad von einem Nachfolgeknoten von `n` zu einem Knoten `n'` gibt, so existiert auch kein Pfad von `n` nach `n'`. Die Aussage ist also äquivalent zu der entsprechenden Aussage aus dem vorherigen Rekursionsschritt.

Für die Aussage aus Abbildung 4.4 wird zunächst wieder betrachtet, wie die Worklist und für welche Knoten sich der Zustand verändert hat. Angenommen, für den Knoten `n'` wurde im aktuellen Schritt durch `updateState` eine neue Definition der Variable `V` aus Knoten `n` eingetragen. Dies impliziert, dass `n'` ein Nachfolgeknoten des zuletzt bearbeiteten Knotens `n''` ist und nun in der Worklist enthalten ist, da diese Nachfolgeknoten hinten an die Worklist angehängt werden. Außerdem muss diese Definition dann auch in `n''` sichtbar sein. Nach der Invariante des vorherigen Schrittes, muss es daher einen Teilpfad `p` eines Pfades von `n` nach `n''` geben, der den ersten oder zweiten Spezialfall aus Abbildung 4.3 erfüllt. Der Knoten `n''` wurde aus der Worklist entfernt, sodass nun ein Teilpfad von `p` existiert, der in `n'` endet und einen der beiden Spezialfälle erfüllt (im Extremfall ist dieser Teilpfad durch den Pfad von `n''` nach `n'` gegeben).

Es bleibt nun noch die Betrachtung der Knoten `m`, für die der Zustand nicht geändert wurde. Ist in `m` die Definition aus Knoten `n` nicht sichtbar, obwohl eine Datenabhängigkeit von `m` nach `n` besteht, so ist keine der beiden Spezialfälle aus Abbildung 4.3 erfüllt.
Das bedeutet für den ersten Spezialfall, dass ein Knoten aus dem Pfad von `n` nach `m` in der Worklist enthalten war. Handelte es sich dabei um den zuletzt bearbeiteten Knoten `n''`, so wurde dieser zwar entfernt, jedoch muss die Definition in `n''` sichtbar gewesen sein, sodass sie in alle Nachfolgeknoten von `n''` eingetragen wurden. Somit hat sich der Zustand verändert, weshalb diese Nachfolgeknoten nun in der Worklist enthalten sind. Da auch ein Nachfolgeknoten von `n''` auf dem Pfad von `n` nach `m` liegen muss und diese alle in der Worklist stehen, ist der erste Spezialfall weiterhin nicht erfüllt.

Für den zweiten Spezialfall bedeutet dies, dass in der Worklist kein Knoten existiert, der die Eigenschaften aus dem Spezialfall erfüllt. Die Knoten, die neu in die Worklist aufgenommen werden, können die Eigenschaften auch nicht erfüllen, da sonst ihr Vorgängerknoten `n''`, der zuvor in der Worklist enthalten war, diese bereits erfüllt hätte. Es liegt also weiterhin keiner der beiden Spezialfälle vor, womit der Fall, dass in `m` die Definition aus `n` nicht sichtbar ist, abgehandelt ist.

Wenn andernfalls eine Definition aus Knoten `n` in `m` sichtbar ist, so lag eine der beiden Spezialfälle vor. War der zweite Spezialfall erfüllt, so liegt er, für den Teilpfad `p'` ohne seinen ersten Knoten, nach einem Rekursionsschritt immer noch vor. Wenn der erste Spezialfall galt und `n` nicht in der Worklist enthalten ist, so ist er nun weiterhin erfüllt. Ist der Knoten `n` nun in der Worklist enthalten, so ist jetzt der zweite Spezialfall eingetreten. Dafür kann der Pfad von einem Nachfolgeknoten von `n`, der selbst nicht in der Worklist ist, nach `m` gewählt werden.

Die restlichen Aussagen der Invariante lassen sich leicht über eine Regelinduktion nach der Funktion `updateState` zeigen.

4.2.3 Herleitung der Hauptaussage aus der Invariante

Es bleibt nun noch zu zeigen, dass sich die Hauptaussage aus der Invariante folgern lässt, wenn die Abbruchbedingung erfüllt ist. Dies bedeutet, dass die Worklist leer ist und somit `set ws` die leere Menge ist. Damit gelten für jeden Knoten `n` die Aussagen $\texttt{n} \notin \texttt{set ws}$ und $\forall \texttt{n'} \in \texttt{set ws}.\ \neg(\exists \texttt{as}.\ \texttt{n'}\ \texttt{-as}\rightarrow^*\ \texttt{n})$. Mit der zweiten Aussage der Invariante aus Abbildung 4.2 folgt somit `Mapping.lookup s n` $\neq$ `None`, das heisst jeder Knoten im CFG wurde besucht.

Für jeden Knoten gilt somit die Äquivalenz aus Abbildung 4.4. Da `set ws = {}` gilt, fallen die beiden Spezialfälle weg und es bleibt somit

```
∀n ∈ getNodes g. ∀V. ∀n' ∈ getNodes g.
  (∃vd ns. Mapping.lookup s n' = Some vd
    ∧ Mapping.lookup vd V = Some ns ∧ n ∈ set ns)
⟷
  (V ∈ Def n ∧
    ∃a as. (n -a#as→* n' ∧ ∀n'' ∈ set (sourcenodes as). V ∉ Def n''))
```

übrig. Die rechte Seite der Äquivalenz ist genau die in Kapitel 2.2 eingeführte formale Definition für Datenabhängigkeit, sodass, zusammen mit der Äquivalenz aus Abbildung 4.1, schliesslich die Hauptaussage folgt.

4.3 Termination des Algorithmus

Um zu zeigen, dass der Algorithmus für beliebige Eingabegraphen terminiert, muss bei der Benutzung des `while_rule_lemma` eine wohlfundierte Relation angegeben werden. In dieser Arbeit wurde dafür eine Maßfunktion benutzt, da solche Funktionen stets wohlfundiert sind. Dafür wurden folgende Funktionen definiert.

```
numDefs :: "'node dd_visibleDefs ⇒ nat"  where
  "numDefs vd = (∑n ∈ Mapping.keys vd.
                 length (the (Mapping.lookup vd n)) + 1)"

stateKl :: "'node dd_state ⇒ nat" where
  "stateKl s = (∑n ∈ Mapping.keys s.
                 numDefs (the (Mapping.lookup s n)) + 1)"

maxDefs :: "'graph ⇒ 'node dd_def ⇒ nat" where
  "maxDefs g d = card (getNodes g) * (∑n ∈ Mapping.keys d.
                 length (the (Mapping.lookup d n)) + 1)"

klMeasure :: "'graph ⇒ 'node dd_def ⇒ 'node dd_state ⇒ nat" where
  "klMeasure g d s = (maxDefs g d) - (stateKl s)"
```

Die Funktion `stateKl` zählt alle im Zustand `s` sichtbaren Definitionen. Für einen gegebenen CFG lässt sich dies, über die Funktion `maxDefs`, nach oben abschätzen, denn jede Definition kann höchstens in jedem Knoten sichtbar sein. Da im Laufe der Rekursion immer mehr sichtbare Definitionen in den Zustand eingetragen werden und keine sichtbare Definition aus dem Zustand entfernt wird, ergibt sich aus der Differenz von `maxDefs` und `stateKl` eine passende Maßfunktion `klMeasure`. Es bleibt zu beachten, dass es auch Rekursionsschritte geben kann, in denen der Zustand und damit auch `stateKl` nicht verändert wird. In diesem Fall nimmt jedoch die Anzahl an Elementen in der Worklist ab, sodass sich die gesuchte Maßfunktion als Kombination von `klMeasure` und der Länge der Worklist ergibt, da `klMeasure` nicht zunehmen kann.

Die wesentlichen Lemmata für diese Aussage sind

$$\texttt{lemma updateState\_increases\_stateKl:}\quad \texttt{stateKl s} \leq \texttt{stateKl (updateState n ds ns s)} \tag{4.1}$$

$$\texttt{lemma updateState\_neq\_inc\_stateKl:}\quad \frac{\texttt{updateState n ds ns s} \neq \texttt{s}}{\texttt{stateKl s} < \texttt{stateKl (updateState n ds ns s)}} \tag{4.2}$$

Das Lemma `updateState_increases_stateKl` besagt, dass die Anzahl der Definitionen durch den Aufruf von `updateState` nicht sinkt, während im Lemma `updateState_neq_inc_stateKl` gezeigt wird, dass eine Änderung des Zustands durch `updateState` stets bedeutet, dass die Anzahl an Definitionen gestiegen ist. Zum Beweis dieser Aussage wurde das Lemma `updateState_neq_cases` bewiesen, das folgende Aussage trifft. Wenn `updateState` den Zustand ändert, so liegt eine der drei folgenden Fälle vor.

1. Im aktuellen Rekursionsschritt wurde ein Knoten zum ersten Mal besucht, das heisst dem Zustand `dd_state` wird ein Knoten als neuer Schlüssel hinzugefügt.

2. In einem bereits besuchten Knoten wird eine neue Variable als sichtbare Definition eingetragen. Dies hat zur Folge, dass in der, dem Knoten zugewiesenen, sichtbaren Definition `dd_visibleDefs` ein neuer Schlüssel eingetragen wird.

3. Eine Variable ist bereits im Knoten als sichtbare Definition eingetragen, allerdings wird die Variable ebenso in einem anderem Knoten definiert und erreicht den aktuellen Knoten, sodass die Liste in der entsprechenden sichtbaren Definition `dd_visibleDefs` erweitert wird.

Jeder dieser drei Fälle hat zur Folge, dass sich `stateKl` erhöht. Das heisst, falls `updateState n ds ns s` $\neq$ `s` gilt, wächst `stateKl`, was eine Verkleinerung von `klMeasure` impliziert. Verändert `updateState` den Zustand jedoch nicht, so bleibt `klMeasure` zwar gleich, die Worklist wird jedoch verkleinert. Also verringert sich in einem Rekursionsschritt entweder der Wert von `klMeasure` oder die Länge der Worklist, sodass die Rekursion nach endlich vielen Schritten terminieren muss.

Somit wurden alle Annahmen für das `while_rule_lemma` gezeigt, womit insgesamt die Hauptaussage und damit die Korrektheit und Vollständigkeit des Algorithmus bewiesen ist.

5 Interpretation und Ausführbarkeit der Formalisierung

Der in dieser Arbeit implementierte Algorithmus wurde verwendet, um Datenabhängigkeiten für eine While Sprache zu berechnen. Dafür liefert die Bachelorarbeit von Simon Kohlmeyer [8] eine Methode zur Erzeugung eines CFG aus einem Programm der While Sprache.

Um aus den in Kapitel 2.5 eingeführten `graph` Locales konkrete Graphen zu erhalten, wurde in der Arbeit von Kohlmeyer eine Instanziierung der Locales durchgeführt, die Rot-Schwarz-Bäume (RBT) als zugrunde liegende Datenstruktur verwendet. Für die Parameter α`e`, `invar`, `empty`, `addEdge` und `outEdges` wurden dafür entsprechende Funktionen `mg_`α`e`, `mg_invar`, `mg_empty`, `mg_addEdge` und `mg_outEdges` definiert.

Ein CFG der While Sprache ist durch die `WhileGraph` Locale definiert, die die `graph` Locales erweitern. Die Instanziierung der While-Graphen verwendet daher ebenfalls RBT als Datenstruktur. Basierend auf den Funktionen `mg_`α`e`, `mg_invar` und `mg_outEdges`, wurden die entsprechenden Funktionen `while_edges`, `while_invar` sowie `while_outEdges` für While-Graphen definiert. Diese wurden zur Instanziierung der `DataDependencyGraph` Locale verwendet, in der die Datenabhängigkeitsberechnung definiert ist. Dabei bezeichnet `(_Entry_)` den Startknoten eines While-Graphen.

interpretation *dd_while: DataDependencyGraph while_edges while_invar while_outEdges mg_*α*e mg_invar mg_empty mg_addEdge "*λ*c. (_Entry_)"*

Die von der `DataDependencyGraph` Locale geforderte Annahme nach der Endlichkeit der Knoten im Graph ist, durch die Verwendung von RBT als Datenstruktur, erfüllt und lässt sich einfach zeigen. Mit Hilfe dieser Instanziierung wurde der Algorithmus auf einigen Beispielprogrammen angewendet und über den Isabelle Codegenerator nach Haskell exportiert. Zusätzlich wurde eine Instanziierung für beliebige, auf RBT basierenden, Graphen erstellt. Die Funktion `Entry` liefert den definierten Startknoten eines Graphen.

interpretation *ddg: DataDependencyGraph mg_*α*e mg_invar mg_outEdges mg_*α*e mg_invar mg_empty mg_addEdge Entry*

Dies ermöglicht es, den Algorithmus auf beliebig aufgebauten Graphen, die nicht zwangsläufig mit einer Sprache in Verbindung stehen, auszuführen. Zu Testzwecken wurde der Algorithmus auf einigen, manuell erstellten, Graphen ausgeführt.

6 Fazit

Das Ziel dieser Arbeit war die Implementierung und Verifikation eines Algorithmus zur Berechnung von Datenabhängigkeiten in einem CFG. Es liegt nun eine ausführbare Berechnung vor, die Datenabhängigkeiten für einen CFG berechnen kann. Seine Korrektheit und Vollständigkeit wurde mit Isabelle verifiziert, sodass dieses Ziel erreicht ist. Weiterhin wurde der Algorithmus erfolgreich auf der, in der Arbeit von Kohlmeyer [8] vorgestellten, While Sprache ausgeführt. Dies ist nicht nur eine konkrete Anwendung des Algorithmus, sondern zeigt auch, dass die Berechnung auf Ergebnisse aus anderen Arbeiten anwendbar ist.

Jedoch wurde in der Arbeit keine aussagekräftige Evaluation bezüglich der Laufzeit des Algorithmus durchgeführt. Es ist daher noch unklar, ob der Algorithmus unter realistischen Bedingungen in angemessener Zeit terminiert. Ebenso wurden Aspekte der Effizienz für den Algorithmus nicht betrachtet, sodass es möglich ist, dass eine effizientere Version des Algorithmus umgesetzt werden kann.

6.1 Ausblick

Um sicherzustellen, dass der Algorithmus bei realistischen Eingaben in angemessener Zeit terminiert, sollte die Laufzeit des Algorithmus evaluiert werden, um eine Anwendung in der Praxis sicherstellen zu können. Abgesehen davon, kann die in dieser Arbeit vorgestellte Berechnung von Datenabhängigkeiten zum Beispiel in einem Slicer, wie er im Projekt *Quis-Custodiet* entwickelt werden soll, verwendet werden. Bislang können nur Programme der While Sprache aus Kohlmeyers Bachelorarbeit [8] in CFG transformiert werden. Für mächtigere Sprachen fehlen noch verifizierte Algorithmen.

Maximilian Wagner implementiert in seiner Bachelorarbeit [12] eine verifizierte Berechnung von Kontrollabhängigkeiten. Wird diese Berechnung mit dem Algorithmus aus dieser Arbeit kombiniert, so wäre es möglich aus einem CFG den Programmabhängigkeitsgraph zu berechnen. An dieser Stelle fehlt eine Möglichkeit Programmabhängigkeitsgraphen zu einem Systemabhängigkeitsgraphen ohne Summary Kanten zusammenzuführen. Die Bachelorarbeit von Altmayer liefert dann einen Algorithmus, der die Summary Kanten berechnet, sodass die nötigen Voraussetzungen für einen Slicer, der ebenso noch zu erstellen ist, vorliegen.

6.2 Verwandte Arbeiten

In der Dissertation von Daniel Wasserrab [13] wird die formale Korrektheit von Slicing mit Isabelle/HOL bewiesen, das die, in dieser Arbeit benutzte, Definition von Datenabhängigkeiten verwendet. Dazu führt Wasserrab ein Slicing Framework ein, das mit verschiedenen Sprachen instanziiert werden kann. Aus diesem Framework lässt sich allerdings kein ausführbarer Code generieren.

Es existieren mehrere Arbeiten, die eine verifizierte Datenflussanalyse durchführen. Ein Beispiel dafür ist der Compiler *CompCert* von Xavier Leroy [9]. Dabei handelt es sich um einen Compiler, der eine große Teilmenge von C in PowerPC Assembler Code übersetzt und mit dem Theorembeweiser Coq verifiziert wird. Im Rahmen dieser Verifikation wird unter anderem die Korrektheit der Datenflussanalyse, die der Compiler zu Optimierungszwecken durchführt, bewiesen. Gerwin Klein und Tobias Nipkow liefern in ihrer Arbeit [7] eine Formalisierung der zu Java ähnlichen Sprache Jinja, einer virtuellen Maschine zur Ausführung von Jinja und eines Jinja Compilers. Ein Teil dieser Arbeit ist die Verifikation der Datenflussanalyse für die virtuelle Maschine mit Isabelle/HOL. Die Formalisierung aus dieser Arbeit ist ebenfalls ausführbar.

Literatur

[1] Altmayer, S. (2012). Verifizierte Summary Kanten Berechnung in System Dependence Graphs. Bachelorarbeit, Karlsruher Institut für Technologie, Fakultät für Informatik.

[2] Ballarin, C. (2004). Locales and locale expressions in Isabelle/Isar. *Types for Proofs and Programs: International Workshop, LNCS 3085*. Springer.

[3] Ferrante, J., Ottenstein, K. J. und Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3).

[4] Güting, R. H. und Erwig, M. (1999). Übersetzerbau - Techniken, Werkzeuge, Anwendungen. Springer.

[5] Hammer, C. und Snelting, G. (2009). Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6).

[6] Huffman, B., Kuncar, O. (2012). Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. New York, NY, USA.

[7] Klein, G. und Nipkow, T. (2006). A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*, 28(4).

[8] Kohlmeyer, K.-S. (2012). Funktionale Konstruktion und Verifikation von Kontrollflussgraphen. Bachelorarbeit, Karlsruher Institut für Technologie, Fakultät für Informatik.

[9] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7).

[10] Nielson, F. und Nielson, H. und Hankin, C. (1999). Principles of Program Analysis. Springer-Verlag.

[11] Nipkow, T., Paulson, L. C. und Wenzel, M. (2013). Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Springer-Verlag.

[12] Wagner, M. (2013). Verifizierte Berechnung von Kontrollabhängigkeiten.

Bachelorarbeit, Karlsruher Institut für Technologie, Fakultät für Informatik.

[13] Wasserab, D. (2010). From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security. Dissertation, Karlsruher Institut für Technologie, Fakultät für Informatik.

[14] Weiser, M. (1979). Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Dissertation, Universität Michigan.

Erklärung

Hiermit versichere ich, Haoqian Zheng, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der aktuell gültigen Fassung beachtet habe.

Karlsruhe, den 15. November 2013

Haoqian Zheng