Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

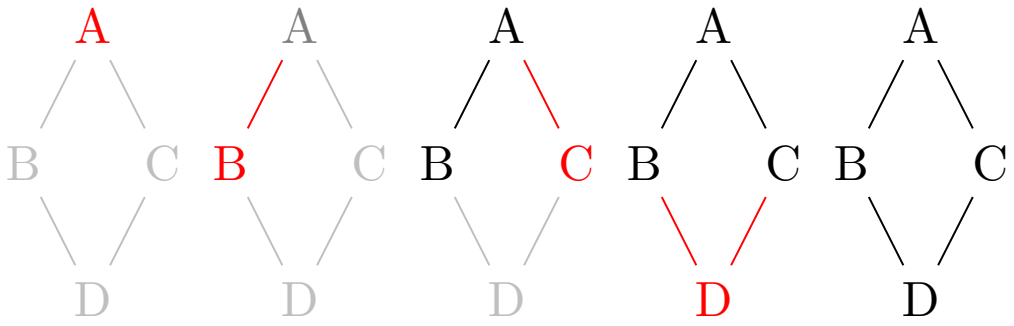
Lehrstuhl Prof. Dr.-Ing. Snelting

Strategien für Datenflussanalysen auf Steuerflussgraphen

Diplomarbeit von

Fabian Sperber

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter: Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 13. Mai 2013 – 12. Februar 2014

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 12. Februar 2014

Inhaltsverzeichnis

1 Einführung

Datenflussanalysen sind ein wichtiger Bestandteil im Prozess der Softwareentwicklung. Zum einen sind sie eine grundlegende Technik, die von Optimierungen in einem Compiler eingesetzt werden, um herauszufinden, an welchen Stellen und wie optimiert werden kann. Ein anderer Gesichtspunkt sind Analysen in einer Programmierumgebung, welche den Benutzer durch z.B. automatischer Vervollständigung, Anzeigen von Informationen verwendeter Variablen oder Methoden oder Erkennen von Programmierfehlern unterstützen können. Ein weiteres großes Gebiet, indem Datenflussanalsen eingesetzt werden, ist in der Programmverifikation. Hier müssen bestimmte Eigenschaften eines Programmes bewiesen werden. Während das reine Compilieren im Vergleich zu der Häufigkeit des Ausführens des entstandenen Binärprogramms üblicherweise verhältnismäßig selten vorkommt, spielte die Dauer bisher meist eher eine untergeordnete Rolle. Jedoch ist für Techniken wie Just-in-time Kompilieren die Laufzeit zeitkritisch für das Reaktionsverhalten des Programmes. Ansätze wie Test Driven Development [3] sind wiederum darauf ausgelegt, den entstehenden Code quasi im Minutentakt zu compilieren. Wenn die verwendeten Datenflussanalysen schneller ausgeführt werden, kann der Benutzer in all diesen Anwendungsfällen durch kürzere Wartezeiten beziehungsweise bessere Reaktionszeiten profitieren. Auch wäre es dadurch möglich, mehr Analysen in der gleichen Zeit durchzuführen, um so bessere Ergebnisse zu erzielen.

Die Datenflussprobleme, die bei diesen Analysen anfallen, werden häufig mithilfe eines iterativen Fixpunkt-Algorithmus gelöst. Für den Algorithmus wird in seiner ursprünglichen Form zwar ein Kriterium angegeben, wann eine Lösung gefunden wurde, die Reihenfolge, in der über den Datenflussgraph iteriert werden soll, ist jedoch nicht definiert. Für die Korrektheit des Algorithmus ist die Reihenfolge unbedeutend, es muss nur sichergestellt werden, dass jeder Knoten betrachtet wird. Der Algorithmus ist einfach zu implementieren und unter bestimmten Beschränkungen kann eine obere Schranke für die Laufzeit angegeben werden [21], was ihn sehr attraktiv macht. In der Praxis gibt es jedoch oft Situationen, bei denen die Analyse nicht innerhalb dieser Beschränkungen liegt, wodurch auch nicht mehr diese obere Schranke gilt. Wie Cooper et al. herausgefunden haben [13], kann die Iterationsreihenfolge eine große Rolle spielen. Sie haben gegenüber dem

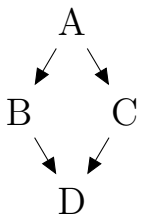


Abbildung 1.1

klassischen iterativen Round-Robin Algorithmus ein Geschwindigkeitsvorteil von 25% bis 40% nachgewiesen.

Ein kleines Beispiel zeigt in Abbildung 1.1, dass die Reihenfolge einen großen Unterschied in der Geschwindigkeit ausmachen kann. Man stelle sich eine Folgen von Operationen an den Stellen A – D vor. Wenn man von A aus rekursiv die Nachfolger besucht, bekommt man die Reihenfolge A–B–D–C–D oder A–C–D–B–D. Hier muss D zweimal betrachtet werden, da es potentiell Änderungen der gefundenen Informationen im Vorgänger gibt. Wenn man in diesem Beispiel stattdessen nach der Breitensuche vorgeht, bekommt man eine kürzere Reihenfolge A–C–B–D. Dies wirft die Frage auf, welche Iterationsstrategie optimal ist bzw. unter welchen Umständen optimal ist.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist die Analyse verschiedener Iterationsstrategien auf ihre Effizienz. Da es keine genauen Abschätzungen für verschiedene Iterationsstrategien gibt, werden in dieser Arbeit einige ausgewählte Strategien empirisch miteinander verglichen. Außerdem wurde den Fragen nachgegangen, ob es eine Iterationsstrategie gibt, die für alle Probleme gut geeignet ist, und falls nicht, welche Strategie für welche Analyse geeignet ist.

Eine damit verbundene Aufgabe dieser Arbeit war es, ein Framework zu entwickeln und in libFirm zu implementieren, mit dem Datenflussanalysen leichter zu implementieren sind. Mit diesem Framework lassen sich verschiedene Iterationsstrategien mit minimalem Aufwand austauschen und vergleichen. Das Framework soll dann verwendet werden, um die verschiedenen Iterationsstrategien für einige Datenflussanalysen zu vergleichen.

1.2 Aufbau

Nach dieser kurzen Einleitung werden in Kapitel 2 die Grundlagen behandelt. Insbesondere werden das in dieser Arbeit verwendete Programm und die dazugehörige Programmbibliothek beschrieben. Kapitel 3 stellt Ansätze anderer Arbeiten vor, die mit diesem Thema verwandt sind und vergleicht diese. In Kapitel 4 wird die Implementierung vorgestellt. Die Ergebnisse werden in Kapitel 5 dargestellt. Abschließend fasst Kapitel 6 zusammen und gibt einen Ausblick auf weitere mögliche zukünftige Arbeiten und Erweiterungen.

2 Grundlagen

In diesem Kapitel werden erst die grundlegende Definition in Abschnitt 2.1 erklärt. Dann wird in Abschnitt 2.2 die verwendete Zwischensprache vorgestellt. In Abschnitt 2.3 werden allgemeine Grundlagen der Datenflussanalyse vorgestellt, während in Abschnitt 2.4 auf die in dieser Arbeit verwendeten Datenflussanalysen eingegangen wird. In Abschnitt 2.5 werden das verwendete Programm cparser und die verwendete Programmbibliothek libFirm vorgestellt.

2.1 Definitionen

Graph

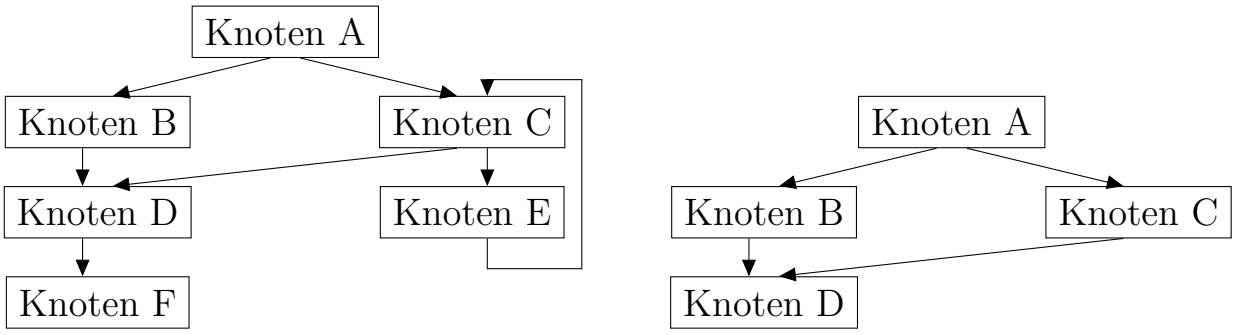
Definition (Graph)**.** *Ein* Graph $G = (V, E)$ *ist eine abstrakte Struktur, die die Menge von Objekten* V*, genannt* Knoten*, sowie zwischen den Objekten bestehende Verbindungen* E*, genannt* Kanten*, repräsentiert. Eine Kante* (a,b) *führt von Knoten* a *zu Knoten* b*. Man schreibt dafür auch* $a \to b$*.*

Wenn für jede Kante $(a,b) \in E$ *auch* $(b,a) \in E$*, dann spricht man von einem* ungerichteten *Graphen, ansonsten von einem* gerichtetem *Graphen.*

Wenn in dieser Arbeit von einem Graphen die Rede ist, ist damit, wenn nicht explizit anders definiert, ein gerichteter Graph gemeint. Beispiele für gerichtete Graphen werden in Abbildung 2.1 gezeigt.

Definition (Teilgraph)**.** *Ein Graph* $G_1 = (V_1,E_1)$ *heißt* Teilgraph *von* $G_2 = (V_2,E_2)$*, falls* V_1 *Teilmenge von* V_2*, also* $V_1 \subseteq V_2$ *und* E_1 *Teilmenge von* E_2 *ist, also* $E_1 \subseteq E_2$*.*

Definition (Induzierter Teilgraph)**.** *Sei* $G_1 = (V_1,E_1)$ *ein Teilgraph von* $G_2 = (V_2,E_2)$*. Gilt zusätzlich, dass* G_1 *alle Kanten zwischen den Knoten in* V_1 *enthält,*



(a) Gerichteter Graph (b) Induzierter Teilgraph

Abbildung 2.1: Beispiel gerichteter Graph und induzierter Teilgraph

die auch in G_2 vorhanden sind, nennt man G_1 einen induzierten Teilgraph von G_2. Man schreibt dafür $G_2[V_1]$.

In Abbildung 2.1 ist auf der linken Seite ein Graph G_1 zu sehen. Auf der rechten Seite ist dazu der induzierte Teilgraph $G_1[\{A,B,C,D\}]$ abgebildet.

Definition (Pfad)**.** *Ein* Pfad *ist eine Folge von Knoten, in der jeweils zwei aufeinander folgende Knoten durch eine Kante verbunden sind. Die Folge $p = \{x_1, x_2, \ldots, x_n\}$ ist ein Pfad, wenn die Kanten $\{(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)\}$ im Graph enthalten sind. Wenn ein Pfad $\{a, \ldots, b\}$ von a nach b existiert, schreibt man dafür $a \rightarrow^* b$. Man sagt auch, b ist von a aus erreichbar.*

Definition (Wurzel)**.** *Ein* Handle *des Graphens $G = (V, E)$ ist eine Menge $H \subseteq V$, so dass es für jeden Knoten $v \in V$ einen Knoten $h \in H$ gibt für den es einen Pfad $h \rightarrow^* v$ gibt. In einem Graph $G = (V, E)$ heißt ein Knoten $n \in V$* Wurzel*, wenn es für jeden Knoten $v \in V$ einen Pfad $n \rightarrow^* v$ gibt. Wenn das Handle nur aus einem Element besteht, entspricht dieses der Wurzel.*

Definition (stark zusammenhängend)**.** *In einem Graphen G heißen zwei Knoten n_1 und n_2 heißen* stark zusammenhängend*, wenn es einen Pfad von n_1 und n_2 und einen Pfad von n_2 und n_1 gibt. Wir definieren $\mathcal{SC} \subseteq N \times N$ als binäre Relation wie folgt:*

$$\mathcal{SC} = \{(n_1,n_2) \mid n_1 \text{ und } n_2 \text{ sind stark zusammenhängend}\}$$

In der Literatur gibt es auch die Definition einer schwachen Zusammenhängigkeit. In dieser Arbeit kann das Wort „stark“ aus starke Zusammenhängigkeit weggelassen werden, da wir immer nur die starke Zusammenhängigkeit meinen.

Man kann zeigen, dass $\mathcal{SC}$ eine Äquivalenzrelation ist. Die Äquivalenzklassen von $\mathcal{SC}$ werden *starke Zusammenhangskomponenten* von G genannt. Wenn der Graph

G genau eine Zusammenhangskomponente enthält, spricht man von einem stark zusammenhängenden Graph.

Die Verbindungen zwischen Zusammenhangskomponenten können als reduzierter Graph dargestellt werden. Jede Zusammenhangskomponente wird durch einen Knoten des reduzierten Graphs dargestellt und es gibt eine Kante zwischen zwei Zusammenhangskomponenten genau dann, wenn es eine Kante von einem Knoten aus der ersten Zusammenhangskomponente zu einem Knoten aus der zweiten Zusammenhangskomponente gibt. Dieser reduzierte Graph ist schleifenfrei.

Definition (Ablaufgraph)**.** *Ein* Ablaufgraph (V, E, s, e) *ist ein Graph* (V, E)*, dessen Knoten den Anweisungen eines Programmes und die Kanten dem Steuerfluss entsprechen. Der Graph enthält außerdem zwei spezielle Knoten: den* Startknoten $s \in V$ *und den* Endknoten $e \in V$*. Diese beiden Knoten sind mit dem Rest des Graphen wie folgt verbunden: Kanten vom Startknoten zu allen möglichen Einsprungspunkten, sowie vom allen möglichen Programmendpunkten zum Endknoten.*

In dieser Arbeit werden, wenn es um den Ablaufgraph allgemein geht, die Knoten als Grundblöcke, also Anweisungen, die ohne Sprünge hintereinander ausgeführt werden können, aufgefasst. Die Kanten entsprechen dabei dem Steuerfluss. Wenn es um libFirm Programme geht, entspricht jeder Knoten einer einzelnen Anweisung und die Kanten entsprechen je nach Knotentyp dem Steuerfluss oder dem Datenfluss.

Definition (Reverse Post-Order)**.** *Tiefensuche ist ein Algorithmus zum Suchen von Knoten in einem Graphen und gibt damit eine Reihenfolge zum Durchlaufen des Graphen an. Die* post-order *(PO) ist die Reihenfolge in der die Knoten bei der Tiefensuche zuletzt besichtigt wurde. Die* reverse post-order *(RPO) ist die umgekehrte Reihenfolge der post-order.*

Verband

Definition (Halbordnung)**.** *Eine binäre Relation* $\leq$ *auf einer Menge* M *heißt* Halbordnung*, wenn sie reflexiv, transitiv und antisymmetrisch ist. Es gilt also:*

$$\forall x \in M : x \leq x \qquad (reflexiv)$$
$$\forall x,y,z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z \qquad (transitiv)$$
$$\forall x,y \in M : x \leq y \wedge y \leq x \Rightarrow x = y \qquad (antisymmetrisch)$$

Definition (Totale Halbordnung)**.** *Eine Halbordnung heißt* total*, wenn alle Elemente vergleichbar sind.*

$$\forall x,y \in M : x \leq y \vee y \leq x$$

Als einfachere Schreibweise sei $x < y \Leftrightarrow x \leq y \wedge x \neq y$ definiert.

Definition (Schranken)**.** *Für eine Teilmenge L einer Halbordnung $(M; \leq)$, $L \subseteq M$, sei $z \in M$ eine* obere Schranke*, wenn für alle $x \in L : x \leq z$ gilt. z heißt* kleinste obere Schranke *oder* Supremum*, wenn jede andere obere Schranke größer ist. Man schreibt dafür $\sqcup L$ (*join*).*

$$(\forall o \in M : (\forall x \in L : x \leq o) \Rightarrow z \leq o \Leftrightarrow z = \sqcup(L)$$

Als vereinfachte Schreibweise für die zwei elementige Menge $K = \{x, y\}$ sei $x \sqcup y = \sqcup K$ definiert. Analog wird die *untere Schranke*, sowie die *größte untere Schranke*, genannt *Infimum*, definiert. Für diese schreibt man $\sqcap L$ und $x \sqcap y$ (*meet*).

Definition (Halbverband)**.** *Eine totale Halbordnung $(M; \leq)$ heißt* Halbverband *$(M; \leq; \sqcup)$, wenn für jede Teilmenge $S \subseteq M$ stets $\sqcup S$ existiert. Ferner gibt es ein größtes Element $\top = \sqcup M$.*

Definition (Verband)**.** *Wenn in einem Halbverband $(M; \leq; \sqcup)$ zu jeder Teilmenge $S \subseteq M$ stets auch $\sqcap S$ existiert, nennt man diese Struktur einen* Verband *$(M; \leq; \sqcup; \sqcap)$. Es gibt dann auch ein kleinstes Element $\bot = \sqcap M$.*

In Verbänden gilt $x \leq y \Leftrightarrow x \sqcup y = y \Leftrightarrow x \sqcap y = x$.

Das kartesische Produkt mehrerer Verbände ist ein Verband.

Ein *Hassediagramm* ist eine graphische Darstellung einer Halbordnung. Eine Aufwärtskante $x \to y$ bedeutet $x < y$, wegen Transitivität bedeutet ein Aufwärtspfad $x \to^* y$ also $x \leq y$. In Abbildung 2.2 wird als Beispiel der Potenzmengenverband $P(\{0,1,2\})$ gezeigt.

Funktionen

Für eine Funktion $f : M_1 \to M_2$ zwischen Verbänden $M_1 = (M_1, \leq_1, \sqcup_1, \sqcap_1)$ und $M_2 = (M_2, \leq_2, \sqcup_2, \sqcap_2)$ gilt:

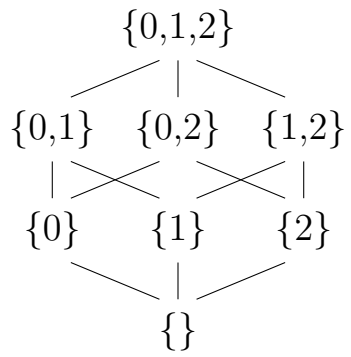


Abbildung 2.2: Hassediagramm des Potenzmengenverbandes $P(\{0,1,2\})$ mit $\subseteq$

Definition (Monotonie)**.** *Die Funktion f ist* monoton*, wenn*

$$\forall m,m' \in M_1 : m \leq_1 m' \Rightarrow f(m) \leq_2 f(m')$$

gilt.

Definition (Distributivität)**.** *Die Funktion f heißt* distributiv*, wenn*

$$\forall m_1,m_2 \in M_1 : f(m_1 \sqcup_1 m_2) = f(m_1) \sqcup_2 f(m_2)$$

gilt.

Definition (Fixpunkt)**.** *Ein Wert $v \in M$ ist ein* Fixpunkt *der Funktion $f : M \to M$, genau dann wenn $v = f(v)$.*

Fixpunkt-Theorem *In einem Verband L mit endlicher Höhe hat jede monotone Funktion f einen eindeutigen kleinsten Fixpunkt $fix(f)$, gegeben durch $fix(f) = \bot \sqcup f(\bot) \sqcup f^2(\bot) \sqcup f^3(\bot) \sqcup f^4(\bot) \sqcup \ldots$.*

Mit Hilfe von Fixpunkten können Gleichungssystem folgender Form gelöst werden:

$$\begin{aligned} x_1 &= f_1(x_1, x_2, \ldots, x_n) \\ x_2 &= f_2(x_1, x_2, \ldots, x_n) \\ &\ldots \\ x_n &= f_n(x_1, x_2, \ldots, x_n) \end{aligned}$$

wobei $x_1, x_2, \ldots, x_n \in L$ Variablen und $f_1, f_2, \ldots, f_n : L^n \to L$ monotone Funktionen sind. Ein solches Gleichungssystem hat eine eindeutige kleinste Lösung, die dem Fixpunkt der kombinierten Funktion F entspricht, die wie folgt definiert ist:
$F(x_1, x_2, \ldots, x_n) = (f_1(x_1,x_2, \ldots, x_n), \ldots, f_n(x_1, x_2, \ldots, x_n))$

2.2 Zwischensprache

In einem Compiler wird der Quellcode gewöhnlich in eine Zwischensprache übertragen, damit der Compiler leichter damit arbeiten kann. Ein weiterer Vorteil ist, dass mehrere Frontends[1] die gleiche Zwischensprache verwenden können. Somit werden für m Quellsprachen und n Zielsprachen nur m Frontends und n Backends[2] anstelle von $n * m$ einzelnen Compiler benötigt – eine eindeutige Arbeitseinsparung. Im Folgenden werden Drei-Adress Code und die SSA-Form vorgestellt.

2.2.1 Drei-Adress Code

Bei Drei-Adress Code werden alle Anweisungen mithilfe eines Operators und maximal 2 Operanden dargestellt. Ein Ausdruck in der Quellsprache wie $x + y * z$ kann als Drei-Adress Code wie folgt dargestellt werden

$t_1 = \mathrm{y} * \mathrm{z}\,;$
$t_2 = \mathrm{x} + t_1\,;$

wobei t_1 und t_2 Compiler generierte temporäre Variablen sind und der Wert in t_2 dem des ursprünglichen Ausdruckes entspricht. Diese Aufteilung von Anweisungen mit mehreren Operatoren ist wünschenswert für die Generierung des Zielcodes sowie für die Optimierung des Codes, da das Bearbeiten solcher einzelner Anweisungen weniger komplex ist.

2.2.2 Static Single Assignment (SSA) Form

Die *Static Single Assignment* (SSA) Form ist eine Zwischensprache, die entwickelt wurde, um Datenflußanalysen zu beschleunigen oder erst zu ermöglichen. Zwei Aspekte unterscheiden die SSA-Form von Drei-Adress Code. Der erste ist, dass es für jede Variable genau eine Zuweisung gibt – daher der Name. Dadurch werden Datenabhängigkeiten zwischen Befehlen explizit dargestellt. Abbildung 2.3 zeigt das gleiche Programm in Drei-Adress Code und in SSA-Form. Die Subskripte unterscheiden jede Definitionen der Variablen e und f in der SSA Darstellung.

[1] Frontend: Teil des Compilers, der Programme in der Quellsprache einliest und zur Weiterverarbeitung (Analyse, Optimierungen, Codegenerierung) in ein Programm der Zwischensprache umwandelt

[2] Backend: Teil des Compilers, der aus dem Zwischencode den endgültigen Zielcode erstellt

```
e = a + b;
f = c - e;
e = a - b;
e = c + e;
f = f + e;
```

(a) Drei-Adress Code

e_1 = a + b;
f_1 = c − e_1;
e_2 = a − b;
e_3 = c + e_2;
f_2 = f_1 + e_3;

(b) Static Single Assignment Form

Abbildung 2.3: Programm in Drei-Adress Code und SSA-Form

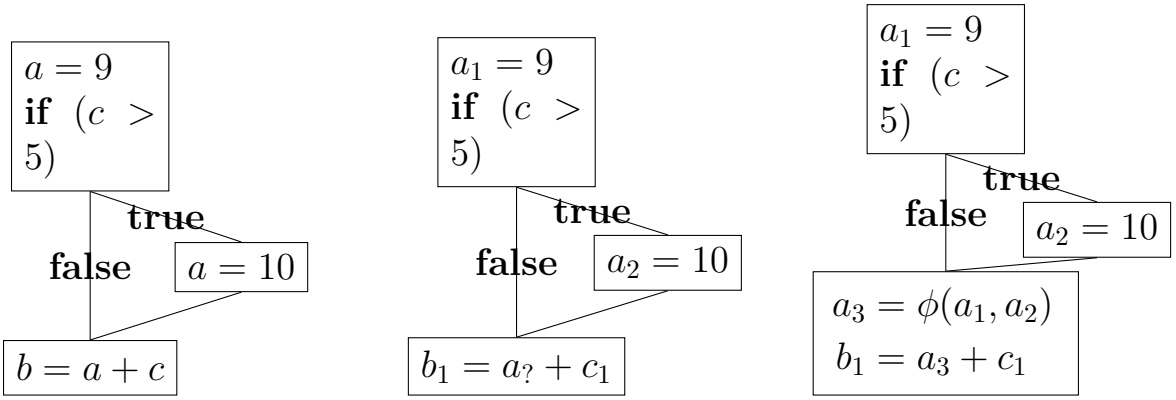
Einer Variablen kann in zwei verschiedenen Steuerflusspfaden ein Wert zugewiesen werden. Das Programmstück

```
if (flag) x = -1; else x = 1;
y = x * a;
```

hat 2 Steuerflußpfade in denen die Variable x definiert wird. Wenn durch die SSA-Form bei jeder Variablenzuweisung eine neue eindeutige Variable verwendet wird, welche Variable soll in der Zuweisung $y = x * a$ verwendet werden? Hier kommt der zweite Aspekt der SSA-Form ins Spiel. Die SSA-Form verwendet eine Notation, genannt ϕ-Funktion, die die beiden Variablen kombiniert:

```
if (flag) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
y = x3 * a;
```

Hier hat $\phi(x_1,x_2)$ den Wert x_1, wenn der Steuerfluß durch den *if* Pfad läuft und den Wert x_2 im *else* Fall. Eine ϕ-Funktion hat genau so viele Operanden wie der zugehörige Grundblock Vorgänger hat. Diese geben den Wert des Argumentes zurück, welches zum Steuerflusspfad gehört, der ausgeführt wurde um zur ϕ-Funktion zu gelangen. Abbildung 2.4 zeigt am Beispiel, wie von Drei-Adress Code zur SSA-Form gelangt werden kann. Hier werden in einem ersten Schritt die Variablenzuweisungen so ersetzt, dass sie nur einmal zugewiesen werden. Im Beispiel wird das durch Hinzufügen von Subskripten erreicht. Im zweiten Schritt werden dann benötigte ϕ-Funktionen hinzugefügt. Für eine vollständige Beschreibung eines Algorithmus zur Berechnung der SSA-Form sei auf die Arbeit von Braun et al. [5] verwiesen.



(a) Drei-Adress Code (b) SSA-Form (im Aufbau) (c) SSA-Form mit ϕ-Knoten

Abbildung 2.4: Beispiel SSA-Form ϕ-Knoten

2.3 Datenflussanalyse

Für Programmoptimierung müssen oft eine bestimmte Klasse von Problemen mit ähnlicher Vorgehensweise gelöst werden. Diese werden *Datenflussproblemen* genannt und aus der Lösung können Informationen über das analysierte Programm gesammelt werden. Kildall beschreibt in [23] eine Klasse an Problemen mithilfe von Verbänden, die mit einem iterativen Algorithmus, der Tiefensuche verwendet, gelöst werden können. In [21] werden die Bedingungen beschrieben, die gelten müssen, damit eine Schranke von $d + 3$ Iterationen verwendet kann. Ausserdem zeigen sie auch, dass der Algorithmus zur Konstantenpropagation von Kildall nicht in diese Klasse fällt.

Programmanalysen geben zur Compile-Zeit sichere Abschätzungen über Werte oder Verhaltensweisen des analysierten Programmes zur Laufzeit. Eine wichtige Anwendung sind Optimierungen im Compiler, um überflüssige Berechnungen zu vermeiden. Darunter fallen z.B. Berechnungen, deren Ergebnisse nicht verwendet werden oder deren Wert zur Compilezeit berechnet werden kann oder Schleifeninvariante Berechnungen, die vor die Schleife bewegt werden können. Neuere Anwendungen der Programmanalysen beinhalten unter anderem das Überprüfen bestimmter Eigenschaften wie das Vorhandensein von Sicherheitslücken.

Die Ausführung eines Programmes kann als Reihe von Transformationen des Programmzustandes, der Menge aller Variablen des Programmes, angesehen werden. Jede Ausführung einer Anweisung ändert den Eingabezustand zu einem neuen Ausgabezustand. Der Eingabezustand entspricht dem Programmzustand vor der Anweisung, der Ausgabezustand dem Programmzustand nach der Anweisung.

Der Programmzustand wird an jeder Programmstelle auf die für die Analyse be-

nötigten Informationen abstrahiert. Diese Datenflußwerte stellen die Information aller möglichen Ausführungspfade, die zu dieser Programmstelle führen können, dar. In komplexeren Analysen müssen Pfade berücksichtigt werden, die zwischen den Steuerflußgraphen mehrerer Funktionen hin- und herspringen. Wir beschränken uns vorerst auf Pfade innerhalb eines Steuerflußgraphens. Interprozedurale Analysen werden in Abschnitt 2.3.1 vorgestellt.

Die Menge aller möglichen Datenflußwerte heißt Dömane der Datenflußanalyse. Für sie wird in Hinblick auf die Implementierung die Eigenschaften eines Verband L mit endlicher Höhe gefordert.

Eine weitere mögliche orthogonale Unterteilung der Datenflussanalysen sind die Kontextsensitivität und Flusssensitivität. Datenflussanalysen lassen sich nach folgenden Kriterien einteilen:

Kontextsensitivität bedeutet, dass im Fall einer interprozeduralen Analyse bei der Analyse des Ziels eines Prozeduraufruf der Aufrufkontext beachtet wird. Im Besonderen bedeutet dies, dass nach Analyse zum entsprechenden Aufruf zurückgesprungen werden kann, während hingegen ohne diese Information die Datenflussinformationen zu allen möglichen Aufrufstellen propagiert werden muss.

Eine Fluss-sensitive Analyse beachtet die Reihenfolge der Anweisungen des Programmes. Eine Fluss-insensitive Zeiger-Alias-Analyse könnte z.B. feststellen, dass Variable x und y auf den gleichen Wert zeigen können – eine Fluss-sensitive Analyse könnte feststellen, dass zwischen Anweisung 30 und 35 die Variablen x und y auf den gleichen Wert zeigen können. Mit dem Thema der Sensitivität setzen sich Marlowe et al. in [26] auseinander. Sie betrachten bisherige Definitionen von Flusssensitivität und unterscheiden diese von der Kontextsensitivität.

Definition Transferfunktion

Eine Menge an monotonen *Transferfunktionen* $f_n : L \to L$ für jeden Knoten n des Steuerflußgraphes. Es wird gefordert, dass es eine Menge F monotoner Funktionen über L gibt, für die folgende Eigenschaften gelten:

- F enthält alle Transferfunktionen f_n
- F enthält die Identitätsfunktion $id(x) = x$

- F ist abgeschlossen unter der Komposition der Funktionen

Eine stärkere Forderung ist die einer distributiven Analyse. Hier müssen die Transferfunktionen zusätzlich auch noch distributiv sein. Dadurch können beim iterativen Fixpunkt Algorithmus bessere Ergebnisse erzielt werden.

Die Datenflußwerte unterliegen Beschränkungen, die durch die Semantik des Programmes festgelegt ist. Diese Beschränkungen werden mit Hilfe der Transferfunktionen beschrieben.

Üblicherweise werden alle Operationen eines Grundblockes zusammengefasst, da dort der Steuerfluss linear ist und dadurch der Zustand nach einer Operation immer dem Zustand vor der nächsten Operation entspricht. An den Stellen, an denen die Grundblöcke betreten werden, ist dies nicht immer der Fall, da z.B. ein Block von zwei Stellen betreten werden kann. Der Zustand für die erste Operation eines Grundblockes muss aus den Zuständen aller vorhergehenden Grundblöcke berechnet werden.

Ein üblicher Ansatz ist es, ein Gleichungssystem aufzustellen und die genaueste Lösung dafür zu finden.

Wenn das Verhalten eines Programmes analysiert wird, müssen alle möglichen Pfade, die die Programmausführung nehmen kann, beachtet werden. Aus den möglichen Werten an einem bestimmten Programmpunkt werden die Informationen extrahiert, die für die jeweilige Analyse interessant sind.

Für jede Anweisung s im Programm gibt es zwei Datenflusswerte, $\mathrm{IN}(s)$ und $\mathrm{OUT}(s)$, die den Programmzustand vor und nach der Ausführung der Anweisung darstellen. Es gibt zwei Gruppen von Bedingungen, die für diese Datenflusswerte gelten müssen:
Basierend auf der Semantik der Anweisung (Transferfunktion f_s) und basierend auf dem Steuerfluss. Für jeden Anweisung s werden folgende Gleichungen festgelegt:

$$\begin{aligned} \mathrm{OUT}(s) &= f_s(\mathrm{IN}(s)) \\ \mathrm{IN}(s) &= \sqcap_{A \in pred(s)} \mathrm{OUT}(s) \end{aligned}$$

An den Stellen, an denen eine Anweisung (s_i) auf die nächste (s_{i+1}) folgt, gilt $\mathrm{IN}(s_{i+1}) = \mathrm{OUT}(s_i)$. An den Stellen, an denen der Steuerfluss zusammenläuft, muss das Infimum der jeweiligen Vorgängerwerte berechnet werden.

Eine Datenflussanalyse wird durch folgende Eigenschaften definiert:

```
1 int a = 5;
2 int b = 0;
3 if (a < 10)
4         b = 10;
5 return b;
```

(a) Beispiel-Code

Zeile	IN-Werte		OUT-Werte	
1	$a = \top$	$b = \top$	$a = 5$	$b = \top$
2	$a = 5$	$b = \top$	$a = 5$	$b = 0$
3	$a = 5$	$b = 0$	$a = 5$	$b = 0$
4	$a = 5$	$b = 0$	$a = 5$	$b = 10$
5	$a = 5$	$b = \bot$	$a = 5$	$b = \bot$

(b) Datenflusswerte

Abbildung 2.5: Beispiel Konstantenpropagation

- ein Verband L
- eine Menge Transferfunktionen $F : L \to L$
- eine Flussrichtung, vorwärts oder rückwärts
- eine endliche Menge an Extremalstellen, typischerweise der Start- bzw. Endknoten
- ein Wert $\iota \in L$ als Startwert für die Extremalstellen

Bei der Betrachtung der Algorithmen ist die Flussrichtung unerheblich. Daher wird nur die Vorwärtsrichtung betrachtet – rückwärts gilt analog.

Die Datenflussanalyse sucht nach einer Lösung dieses Gleichungssystems. Seien die Datenflusswerte $\overrightarrow{X}$, dann wird der Fixpunkt $F(\overrightarrow{X}) = \overrightarrow{X}$ gesucht.

Diese Variablen können als Graph angesehen werden und diesen läuft der Iterationsalgorithmus ab.

Am Beispiel der Konstantenpropagation sollen die Variablen und Gleichungen erläutert werden. Der Verband enthält für jede Integer-Variable im Programm einen Wert aus dem Verband in Abbildung 2.9. Der Startwert $\top$ steht für keine Information über die Variable, eine Konstante entspricht dem Wert und $\bot$ steht für zu viel Information, also dass die Variable nicht konstant ist.

```
for i = 1 to N
    initialize node i

while (sets are still changing)
    for i = 1 to N
        recompute node i
```

Abbildung 2.6: Round-robin iterative Datenflussanalyse

2.3.1 Iterative Datenflussanalyse

In Abbildung 2.6 wird der Algorithmus der klassischen Analyse gezeigt. Es wurde bewiesen, dass dieser Algorithmus für bestimmte Teilklassen terminiert und eine Lösung findet, die Da dieser Algorithmus nur besagt, dass solange Knoten des Graphen neuberechnet werden müssen, bis sich nichts mehr ändert, kann einfach gezeigt werden, dass die in dieser Arbeit beschriebenen Algorithmen die gleichen Eigenschaften besitzen. Die nicht vorgegebene Reihenfolge, in der die Knoten berechnet werden, hat keinen Einfluss auf die gefundene Lösung. Wie man allerdings an einfachen Beispielen sehen kann, hat die Reihenfolge jedoch einen Einfluss auf die Geschwindigkeit, mit der der Algorithmus terminiert.

Bei iterativen Datenflussanalysen werden erst alle Knoten mit einem Startwert vorinitialisiert. Danach wird in der Reihenfolge, die durch den ausgewählten Algorithmus vorgegeben ist, für jeden Knoten die Transferfunktion ausgeführt. Dieser Schritt wird solange durchgeführt, bis sich keine Änderungen mehr ergeben. An den Stellen, an denen der Steuerfluss mehrerer Knoten zusammenkommt, muss für die Meetfunktion für die verwendeten Werte ausgeführt werden.

Im Falle der Vorwärtsanalyse entsprechen die Stellen der Meetfunktion den Phi-Knoten.

Damit sichergestellt werden kann, dass der Algorithmus terminiert, werden mehrere Anforderungen gestellt. Die Menge der Datenflusswerte muss endlich sein. Transferfunktion und Join müssen monoton sein. Monotonie garantiert, dass jede Iteration der Wert gleich bleibt oder wächst, die endliche Größe gewährleistet, dass die Werte nicht unendlich steigen. Daher muss eine Situation erreicht werden, in der $F(x) = x$ für alle x ist, also der Fixpunkt.

Lösungen

Ideale Lösung

Betrachten wir den Eintrittspunkt eines Knotens B (IN(B)). Die ideale Lösung verwendet die Ergebnisse aller möglichen Ausführungspfade, die vom Startpunkt des Programmes zum Anfang von B gelangen. Ein Pfad ist genau dann „möglich“, wenn es eine Berechnung gibt, die diesem Pfad folgt. Die ideale Lösung würde dann den Datenflusswert am Ende aller dieser Pfade berechnen und darauf den Meet-Operator anwenden, um das Infimum zu berechnen.

Jeder Wert größer als der Idealwert ist inkorrekt. Jeder Wert kleiner als der Idealwert ist eine konservative Abschätzung, also sicher.

Meet over all Paths (MOP)

Eine Annäherung an die ideale Lösung wird erreicht, indem wir annehmen, dass jeder Ausführungspfad genommen werden kann. Die Pfade, die für bei MOP betrachtet werden, ist eine Obermenge der Pfade, die bei der idealen Lösung betrachtet werden. Daher werden nicht nur die Datenflusswerte aller möglichen Pfade betrachtet, sondern auch Werte, deren Pfade nicht ausgeführt werden können. Es kann keine bessere Lösung als die ideale Lösung gefunden werden.

Die MOP Lösung hat folgende Probleme: potentiell unendlich viele Pfade wegen Schleifen und selbst, wenn die Anzahl der Pfade endlich ist, kann es unpraktikabel sein, da der Aufwand zu hoch ist.

Im Allgemeinen ist die MOP Lösung unentscheidbar für monotone Datenflussanalysen [22].

Maximum Fixedpoint (MFP)

Die MOP Definition eignet sich nicht direkt für einen Algorithmus, da die Anzahl der Pfade möglicherweise unbeschränkt ist, wenn es Zyklen im Graph gibt. Der iterative Algorithmus findet nicht erst alle Pfade, die zu Knoten B führen, um dann das Infimum zu berechnen. Stattdessen besucht der Algorithmus Knoten in einer beliebigen Reihenfolge und berechnet dabei an diesen Stellen das Infimum

der Datenflusswerte, die bisher berechnet worden sind. Der Algorithmus hält erst, wenn sich keine Werte mehr ändern. Dabei werden alle Werte vorher mit dem Startwert $\top$ initialisiert.

Es ist möglich, dass ein Knoten besucht wird, bevor dessen Vorgänger besucht worden sind.

Es wurde in [19] gezeigt, dass MFP eine sichere (konservative) Abschätzung von MOP ist. Im Fall distributiver Transferfunktionen ist die Lösung sogar gleich.

Interprozedurale Analyse

Die bisher betrachteten Datenflussanalysen werden INTRAPROZEDURALE Analysen genannt, da sie nur auf den Informationen einer einzigen Prozedur arbeiten. An Stellen, an denen andere Prozeduren aufgerufen werden, kann die aufgerufene Prozedur verschiedene Nebeneffekte haben wie z.B. das Verändern von globalen Variablen oder auch die Möglichkeit, dass diese Prozedur nicht zurückkehrt, da sie in einer Endlosschleife iteriert. Auch über die Prozedurparameter sind normalerweise keine Informationen verfügbar. Intraprozedural Analysen haben nur wenig Kenntnisse an den Grenzen der Prozeduren und müssen daher den schlimmst möglichen Fall annehmen. Das bedeutet normalerweise keine Information über Variablen, die außerhalb der Prozedur definiert sind. Am Beispiel der Konstantenpropagation bedeutet dies, dass nur die lokalen Variablen betrachtet werden können und alle Prozedurparameter als NICHT KONSTANT angenommen werden.

Analysen, die eben diesen Effekt von Prozeduraufrufen mit beachten und daher auf mehreren Prozeduren bzw. dem ganzen Programm arbeiten, werden INTERPROZEDURALE Analysen genannt. Eine einfache und nützliche Technik ist das *Inlining*, bei dem ein Prozeduraufruf durch den kompletten Prozedurrumpf ersetzt und entsprechend für die übergebenen Parameter und den Rückgabewert angepasst wird. So wird das Problem der interprozeduralen Analyse auf eine intraprozedurale Analyse abgebildet. In dieser Arbeit wird daher nicht weiter auf Inlining eingegangen.

Wie in Abbildung 2.7 zu sehen, kann eine Prozedur von mehreren Programmstellen aufgerufen werden. Der Programmfluss kann z.b. für Prozedur f von S_f dem Pfad über C_i, S_h, E_h, R_f nach E_f folgen. Der gezeigte Graph würde aber auch einen den folgenden Pfad zulassen: von S_f über C_i, S_h, E_h, R_g nach E_g. Dieser zweite Pfad ist kein gültiger Pfad, da der Programmfluss von der Prozedur h

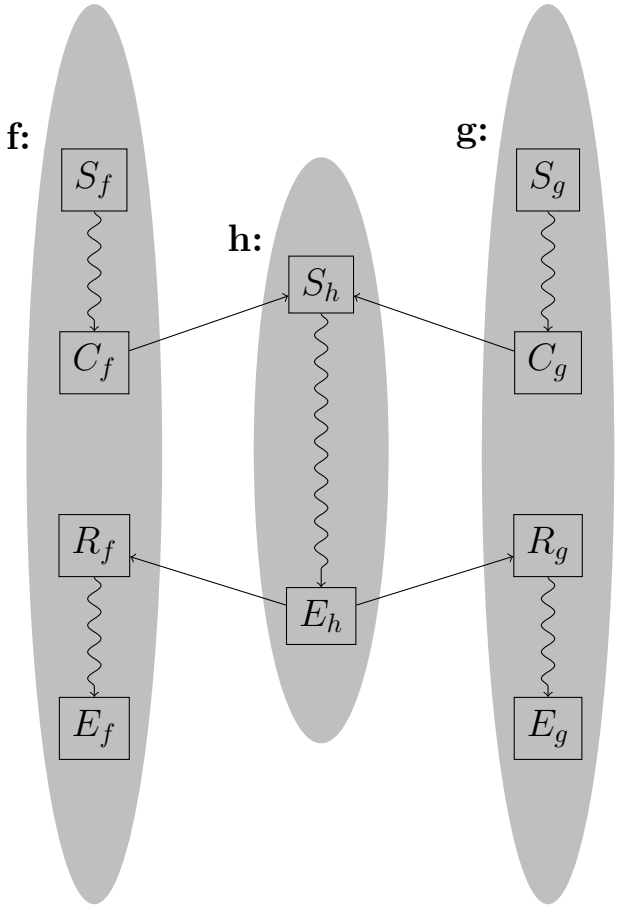


Abbildung 2.7: Beispiel Prozeduraufruf

zu einer Prozedur springt, die diese nicht aufgerufen hat. Hierfür wird der Begriff der *Kontextsensitivität* verwendet. Wenn eine Analyse auch durch Aufrufe und Rücksprünge wie einfache Kanten in einem Graph betrachtet und damit das Ergebnis sowohl einfacher als auch ungenauer macht, spricht man von kontextinsensitiver Analyse. Wenn im Gegensatz sichergestellt werden muss, dass Aufruf und Rücksprung zueinander passen, nennt man eine solche Analyse kontextsensitiv. In dieser Arbeit werden für interprozedurale Analysen nur kontextsensitive betrachtet.

Meet over all Valid Paths (MVP)

Wie gerade in dem Beispiel gezeigt, ist die MOP Lösung bei interprozeduralen Analysen zu ungenau, da dabei Pfade von jedem Rücksprung einer Prozedur zu allen Prozeduraufrufstellen dieser Funktion betrachtet werden. Wenn wir die MOP Lösung so abändern, dass nur Pfade beachtet werden, die korrekt geschachtelte Prozeduraufrufe und -rücksprünge enthalten, gelangen wir zur Definition der Meet over all Valid Paths Lösung.

Maximum Fixedpoint (MFP)

Wie schon bei der intraprozeduralen Analyse die MOP Lösung, ist im interprozeduralen Fall auch die MVP Lösung im allgemeinen unentscheidbar. So müssen wir den MFP Lösungsansatz überdenken, um zu vermeiden, dass zu viele ungültige Pfade betrachtet werden. Dabei müssen Informationen über die besuchten Pfade in die Lösung mit einfließen. Es gibt mehrere Ansätze um die intraprozedurale Analyse als interprozedurale Analyse zu generalisieren.

Eine Möglichkeit Kontextinformationen darzustellen, ist es, sogenannte Call-Strings zu verwenden. Dabei wird der bei der Analyse verfolgte Pfad encodiert, da jedoch nur die Prozeduraufrufe interessant sind, wird der Call-String nur aus diesen aufgebaut. Ein Call-String kann potentiell unendlich lang werden, da Prozeduren rekursiv sein können. Daher wird die Länge des Call-String üblicherweise auf k beschränkt für $k \geq 0$. Die Idee dabei ist, dass die letzten k Aufrufe gemerkt werden.

Eine andere Möglichkeit ist es, die aufgerufenen Prozeduren gedanklich für jeden einzigartigen Kontext zu kopieren. Auf diesem neuen größeren Graph kann dann eine kontextinsensitive Analyse durchgeführt werden. Wenn diese Technik implementiert wird, müssen natürlich nicht ganze Prozeduren kopiert werden, stattdessen kann man mithilfe einer effizienten internen Repräsentation die Ergebnisse der verschiedenen Kopien speichern.

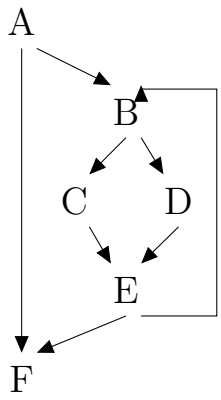
2.4 Verwendete Datenflussanalysen

In diesem Abschnitt werden die in dieser Arbeit verwendeten Datenflussanalysen vorgestellt. Die Analysen werden allgemein beschrieben. Auf Details für die Implementierung in libFirm wird in Abschnitt 4.4 eingegangen.

2.4.1 Postdominanz (PDOM)

In einem Ablaufgraphen (V, E, s, e) mit Endknoten e *postdominiert* ein Knoten $a \in V$ den Knoten $b \in V$, wenn jeder Pfad von Knoten b zum Endknoten e den Knoten a enthält. Man schreibt auch $a \,\text{postdom}\, b$.

$$(\forall p = b \rightarrow^* e : a \in p) \Rightarrow a \,\text{postdom}\, b$$



(a) Ablaufgraph

Knoten	Postdominatoren
A	$\{A,F\}$
B	$\{B,E,F\}$
C	$\{C,E,F\}$
D	$\{D,E,F\}$
E	$\{E,F\}$
F	$\{F\}$

(b) Postdominatoren

Abbildung 2.8: Beispiel Postdominatoren

Die Postdominatoren eines Knoten n kann als maximale Lösung der folgenden Datenflussgleichungen berechnet werden:

$$\begin{aligned} Postdom(e) &= \{e\} \\ Postdom(n) &= (\bigcap_{p \in succ(n)} Postdom(p)) \bigcup \{n\} \end{aligned}$$

Der Postdominator des Endknotens e ist der Endknoten selbst. Die Menge der Postdominatoren jedes anderen Knotens n ist die Schnittmenge aller Postdominatoren der Nachfolger von n. Ausserdem ist n in der Menge der Postdominatoren von n.

Ein Postdominator p des Knotens n heisst *strikter* Postdominator, wenn p ungleich n, also $p \,\mathrm{spostdom}\, n := p \,\mathrm{postdom}\, n \wedge p \neq n$. Ein Postdominator p des Knotens n heisst *direkter* Postdominator, wenn es keinen strikten Postdominator z des Knotens n gibt, der von p strikt postdominiert wird.

$$p = \mathrm{ipostdom}(n) := p \,\mathrm{spostdom}\, n \wedge \neg\exists z : p \,\mathrm{spostdom}\, z \,\mathrm{spostdom}\, n$$

Das Beispiel in Abbildung 2.8 zeigt einen Ablaufgraph und die Postdominatoren aller Knoten.

Cooper et al. zeigen in [11] wie die Algorithmen zur Berechnung der Dominanz mit der Zeit entwickelt wurden. Der Lengauer-Tarjan Algorithmus wird als der meist bekannte und verwendete Algorithmus mit einer fast linearen Zeitschranke angegeben. Die Autoren vergleichen den iterativen Datenflussalgorithmus mit Lengauer-Tarjan und kommen zu dem Schluss, dass, bei geschickter Wahl der Datenstrukturen, der Datenflussalgorithmus trotz schlechterem asymptotischem Verhalten in der Praxis schneller ist.

2.4.2 Konstantenpropagation (CPROP)

Im folgenden wird die Konstantenpropagation informell beschrieben. Für eine detailliertere Beschreibung zeigen Wegman und Zadeck in [33] mehrere Arten der Konstantenpropagation. Wir gehen hier nur auf die *sparse conditional* Konstantenpropagation ein.

Die Konstenpropagation entfernt gleichzeitig einen Teil des toten Codes und ersetzt Werte in Ausdrücken durch Konstanten, falls diese für jede Ausführungsreihenfolge konstant sind. Bei einer Programmverzweigung wird die Bedingung der Verzweigung so gut wie möglich anhand der abstrakten Werte der Variablen ausgewertet. Wenn diese genau, also konstant, sind, kann entschieden werden, welcher Teil der Verzweigung verwendet wird. Wenn die Werte nicht konstant sind, müssen beide (bzw. alle) Verzweigungen betrachtet werden. Nach Durchlauf der Analyse werden alle Anweisungen, die nicht erreicht worden sind, als toter Code markiert und entfernt. Variablen, die als konstant erkannt worden sind, werden an den entsprechenden Verwendungsstellen durch eine Konstante ihres abstrakten Wertes ersetzt.

Die Transferfunktionen werden am Beispiel binärer Operatoren wie folgt definiert:

$$transfer_{op} = \begin{cases} \top & Operand_1 = \top \vee Operand_2 = \top \\ \bot & Operand_1 = \bot \wedge Operand_2 = \bot \\ op(Operand_1, Operand_2) & sonst \end{cases}$$

Unäre oder mehrstellige Operatoren werden entsprechend definiert. Das Ergebnis der Analyse kann verbessert werden, indem bestimmte Eigenschaften von Operatoren genutzt werden. So ist bei der Multiplikation mit 0 das Ergebnis immer 0, auch wenn der anderen Operand $\top$, also auf jeden Fall nicht konstant ist.

Betrachten wir folgenden Code als Beispiel. Zur Verdeutlichung der Arbeitsweise der Konstantenpropagation werden die Werte der Variablen gleich eingesetzt und stellen dabei dann die abtrakten Werte da, mit denen der Algorithmus arbeitet.

```
int x = 6;
int y = 4 * x + 3;
return (y - 7) / 5;
```

Nachdem der Wert von x propagiert wurde:

```
int x = 6;
```

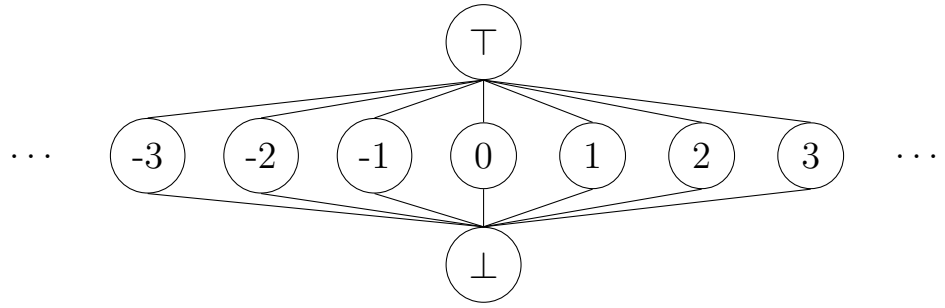


Abbildung 2.9: Verband Konstantenpropagation für eine Variable

```
int y = 27;
if (((y - 7) / 5) > 0) {
  x = 0;
}
return x;
```

Nachdem der Wert von y propagiert wurde:

```
int x = 6;
int y = 27;
if (true) {
        x = 0;
}
return x;
```

Nachdem der Wert von x aus der Programmverzweigung propagiert wurde:

```
int x = 6;
int y = 27;
if (true) {
  x = 0;
}
return 0;
```

Konstantenpropagation ist eine Optimierung, die Werte in Ausdrücken durch Konstanten ersetzt, falls diese zur Compilezeit als konstant erkannt werden können. Die Analyse untersucht jede Anweisung, ob sie für alle Ausführungen ein konstantes Ergebnis hat.

Die Datenflussanalyse verwendet meist flache Verbände als abstrakte Werte für die Variablen des Programmes.

Der Verband, der für die Konstantenpropagation verwendet wird, ist das kar-

tesische Produkt der Verbände aller Variablen des Graphen. Der Verband eines einzelnen Integer-Wertes wird in Abbildung 2.9 dargestellt. Neben $\top$ und $\bot$ ist für jede mögliche Konstante ein Wert vorhanden. Dabei steht $\top$ für den Zustand, in dem die Variable nicht konstant ist, $\bot$ für den Zustand, in dem der Wert der Variablen noch nicht bekannt ist.

2.4.3 Value Range Propagation (VRP)

Bei der Konstantenpropagation werden die Werte der Variablen auf drei mögliche Zustände abstrahiert – unbekannt, konstant, nicht konstant. Dabei geht möglicherweise sinnvolle Information verloren. Gegeben sei folgendes Beispiel.

```
int y = 0;
for (int i=0; i<10; ++i) {
  if (i>100) {
    y = 10;
  }
}
return y;
```

Die Konstantenpropagation kann für den Wert von i nur feststellen, dass dieser nicht konstant ist. Daher muss die Anweisung in der Bedingung für den Wert von y betrachtet werden und dieser wird dadurch auch auf nicht konstant gesetzt. Wenn stattdessen bekannt wäre, dass der Wert von i immer zwischen 0 und 9 ist, könnte erkannt werden, dass die Bedingung innerhalb der Schleife immer nicht wahr ist. Somit wird der abstrakte Wert am Ende des Beispiels von $y = 0$.

Die *Value Range Propagation* (VRP) ist eine Technik um die Schranken der möglichen Wertbereiche von Variablen an bestimmten Programmpunkten zu bestimmen. Sie wurde zu erst von Harrison in [18] beschrieben. Um die Werte von Variablen genau abbilden zu können, müssen die Werte symbolisch dargestellt werden. So soll z.B. dargestellt werden können, dass eine Schleifenvariable einen geraden Integerwert im Bereich von 0 bis 10 hat. Oder dass der Wert einer Variablen abhängig von dem einer anderen ist. Das hat zur Folge, dass Value Range Propagation in dieser Form zwar für Beweise geeignet ist, aber weniger für eine konkrete Implementierung in einem Compiler.

Je nach Verwendung der Analyse kann die Darstellung des Wertebereichs einer Variablen genauer oder ungenauer gewählt werden. Patterson verwendet in [28] als Darstellung eine Menge an Bereichen, wobei jeder Bereich durch eine Wahr-

scheinlichkeit, untere und obere Schranken, sowie einer Schrittgröße angegeben ist. Eine weitere Darstellung implementierte Fietz in [16] für libFirm . Dabei wird zusätzlich zu einem Bereich mit oberer und unterer Schranke auch die gesetzten oder nicht gesetzten Bits einer Variable analysiert.

2.4.4 Don't-care Bits (DCA)

Die *Don't-care Bits* Analyse sucht für Variablen welche Bits für nachfolgenden Berechnungen relevant sind. Diese Analyse wird in libFirm zusammen mit der Value-Range-Propagation-Analyse verwendet, um von Variablen und Operationen verwendete Bits sowie die Konvertierung zwischen Operationen verschiedener Bitgrößen zu reduzieren.

Die Datenflussgleichungen arbeiten mit einem Verband, der für jedes Bit einer Variable einen Wert „cared“ oder „not cared“ besitzt. Als Beispiel sei der Verband für die Bitbreite 3 in Abbildung 2.10 gezeigt.

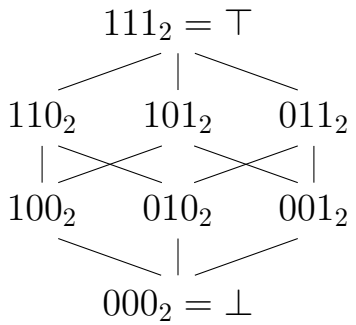


Abbildung 2.10: Hasse Diagramm des DCA-Verbandes für die Bitbreite 3

Die Analyse arbeitet auf dem Rückwärtsgraph. Die Transferfunktionen berechnen, welche Bits eines Knotens möglicherweise verwendet werden. Der Startknoten der Analyse, also der Endknoten des Funktionsgraphen, startet mit dem Wert $\top$, also „alle Bits benötigt“.

2.5 libFirm und cparser

libFirm ist eine Programmbibliothek, die Zwischencode namens Firm , sowie Analysen und Optimierungen darauf bereitstellt. In Firm wird ein Programm mit Hilfe einer Graph-basierten SSA Form dargestellt. Es werden also z.B. keine

Befehlslisten verwendet, sondern nur Datenfluss- und Steuerflussgraphen. Ein Beispiel ist in Abbildung 2.11 zu sehen. Die Variablen werden durch Knoten abgebildet und da durch die Kanten eine Verwendung eindeutig ist, kann auf Variablennamen verzichtet werden. Funktionsparameter werden mit Hilfe von sogenannten *Projektions*-Knoten aus dem Startknoten extrahiert. Im Beispiel sind das „Proj Is Arg 0 59“ für die Variable a und „Proj Is Arg 1 60“ für die Variable b. Operationen verweisen auf ihre Eingabewerte, indem es für jeden Wert eine Kante zum jeweiligen Knoten gibt. Knoten „Add Is 61“ entspricht der Berechnung $a + b$. Für weitere Details sei der Leser auf [6] und [25] verwiesen.

Ein Programm wird in Firm immer in SSA-Form dargestellt. Es gibt nur diese Darstellungsform, daher kann auf Konvertierungsschritte zu anderen Zwischensprachen verzichtet werden. Da libFirm einige Konzepte von Click verwendet, kann der Leser sich in [10] vertiefen.

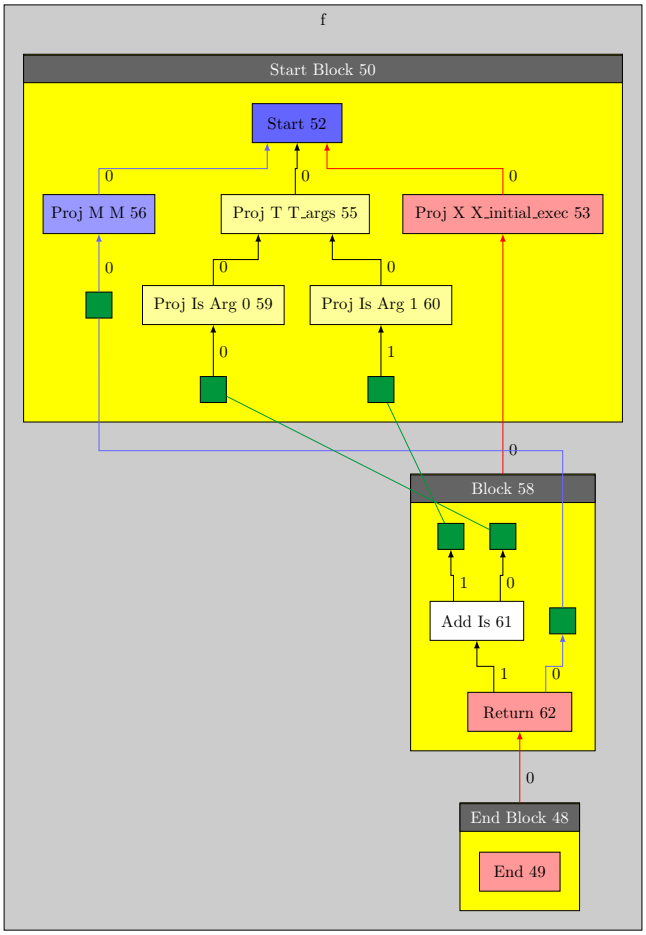
cparser ist ein C Compiler, der libFirm als Implementierung von Firm als Zwischensprache und für Optimierungen verwendet. Der Compiler wird zusammen mit libFirm am Institut für Programmstrukturen und Datenorganisation des KIT entwickelt und wird für verschiedene Forschungsarbeiten verwendet.

2.5.1 Datenflussanalysen in libFirm

Die vorhandenen Datenflussanalysen in libFirm verwenden jeweils ihren eigenen Iterationsalgorithmus. Bei den untersuchten Analysen wurde ein einfacher Worklist-Algorithmus verwendet.

```c
int f(int a, int b)
{
  return a + b;
}
```

(a) C Code



(b) libFirm Graph

Abbildung 2.11: Beispiel libFirm Graph

3 Verwandte Arbeiten

In der Literatur wurden einige Frameworks zur Lösung von Datenflussproblemen beschrieben. `Sharlit` [31] ist ein Analyse- und Optimierungsgenerator, der iterative interprozedurale Analyse unterstützt. Er hat die Möglichkeit Pfade mithilfe festlegbarer Regeln zu komprimieren um so Analysen zu beschleunigen, indem die Transferfunktionen z.B. eines Grundblockes oder größerer Strukturen zusammengefasst werden. `FIAT` [17] ist ein Framework mit dem interprozedurale Analysen definiert werden können. Die interprozedurale Analyse basiert auf intraprozeduraler Analyse für die einzelnen Prozeduren. `FIAT` unterstützt kontext-sensitive Analysen nicht direkt, kann aber dafür erweitert werden. Dazu besitzt `FIAT` eine Schnittstelle um Datenflussprobleme intraprozedural zu beschreiben und das Framework berechnet daraus die Lösung für das interprozedurale Datenflussproblem. Dwyer und Clarke erlauben in ihrem Framework [15] zusätzlich die Definition des Algorithmus zum Lösen des Datenflussproblems. Über die Vorteile wie Code-Wiederverwendung und Erweiterbarkeit haben Adl-Tabatabai et al. in [1] berichtet. Ihr `cmcc` Compiler unterstützt auf Bitvektoren basierende Datenflussanalysen. Chambers et al. beschreiben in ihrer Arbeit [7] die Fortschritte, die sie gegenüber bisherigen Frameworks gemacht haben. Dazu gehören u.a. zusammengesetzte Analysen, bei denen die Analysen so ineinander verschachtelt werden, dass sie quasi parallel ausgeführt werden. Dadurch können bessere Fixpunkte erreicht werden als wenn die Analysen abwechselnd hintereinander ausgeführt werden [9]. Das ROSE Compiler Framework [29] wurde am Lawrence Livermore National Laboratory entwickelt, um schnell und einfach Compilertechniken auf Code anzuwenden. Es bietet ähnlich zu libFirm Analysen und Optimierungen des Codes an, versucht jedoch näher am Code zu arbeiten, wohingegen libFirm näher am Maschinencode arbeitet.

Da diese Arbeit auf der Firm Zwischensprache und damit auf einer SSA-Form arbeitet, können Use-Def Beziehungen direkt verwendet werden. Einige Arbeiten beschreiben das Berechnen der SSA-Form als oft zu aufwändig, jedoch gibt es auch neue Ansätze, die SSA-Form zu berechnen wie Braun et al. in [5] zeigen. Choi et al. beschreiben in [8] eine alternative Möglichkeit, den „*sparse data flow evaluation graph*“. Informationen werden mithilfe des Graphen nicht durch den kompletten Graphen propagiert, sondern nur direkt zu den Stellen, an denen diese

Information gebraucht wird. Die Autoren geben eine Möglichkeit an, mithilfe des sparse data flow evaluation graph die *pruned* SSA-Form zu berechnen, also eine SSA-Form, in der keine ϕ-Funktionen vorkommen, deren Wert nicht verwendet wird.

Hind und Pioli vergleichen in [20] empirisch vier nicht kontextsensitive Pointer Alias-Analysen mit verschiedenen Graden von Flusssensitivität. Sie gehen weiterhin auf die Implementierungstechniken ein und messen deren Geschwindigkeitszuwachs. Dabei zeigen sie, dass ein Worklist Algorithmus schneller als Round-Robin ist, und außerdem der Worklist Algorithmus davon profitieren kann, wenn die Workliste sortiert ist. Eine weitere Verbesserung erreichen sie, indem sie ihren sogenannten ForwardBind Filter auf die Datenflusswerte anwenden. Dieser entfernt bei der Analyse einer aufgerufenen Funktion die Alias-Beziehungen, die bei dieser Funktion nicht betrachtet werden und fügt sie dann direkt zum Ergebnis Datenflusswert wieder hinzu, um so dem Umfang und damit den Aufwand der analysierten Funktion zu verringern. In [32] zeigen Tok et al. einen neuen Algorithmus zur interprozeduralen Analyse, der effizienter als seine Vorgänger ist. Dieser Algorithmus verwendet Datenabhängigkeiten um die Anzahl der ausgeführten Transferfunktionen zu reduzieren. Da der Algorithmus auf Grundblöcken arbeitet, werden die benötigten Def-Use-Ketten beim Ausführen am Anfang des Algorithmus berechnet, um weiteren Aufwand während der Analyse zu sparen. Im Besonderen gehen die Autoren auf die Analyse von Zeigern ein.

Cooper et al. zeigen in [12], dass die Iterationsreihenfolge eine wichtige Rolle für die Laufzeit des Algorithmus spielt, indem sie beim allgemeinen Worklist-Algorithmus die Geschwindigkeit verbessern. Durch verschiedene Datenstrukturen und Reihenfolgen des Hinzufügens in diese ändern sich das Laufzeitverhalten. Als Ergebnis ihrer Arbeit geben die Autoren bei geschickter Wahl des Algorithmus einen Geschwindigkeitszuwachs von bis zu 40% im Vergleich zum round-robin Algorithmus an. Auch wird erwähnt, dass durch unpassend gewählte Parameter des Algorithmus die Geschwindigkeit sehr leicht verschlechtert werden kann. Während in der Arbeit von Cooper et al. gezeigt wird, dass man durch die Wahl des Algorithmus bessere Performanz erlangen kann, wurden in dieser Arbeit weitere Algorithmen untersucht, im Besonderen das Beachten der Zusamenhangskomponenten. Weitergehend wurde untersucht, ob es einen Algorithmus gibt, der für verschiedene Datenflussanalysen immer oder oft bessere Performanz bietet als andere Algorithmen.

Bourdoncle untersucht in [4] effiziente Reihenfolgen für chaotische Iteration. Dabei betrachtet er Verbände mit unendlicher Höhe bzw. sehr großer Höhe, für die Beschleunigungstechniken wie Widening und Narrowing verwendet werden müssen. Er macht auch Angaben über Komplexität und Implementierungsdetails

verschiedener Algorithmen.

Einen ganz anderen Ansatz verfolgen Arzt und Bodden in ihrer Arbeit [2]. Sie gehen davon aus, dass Programmcode gewöhnlich inkrementell erweitert wird. Datenflussanalysen müssen üblicherweise den gesamten Code analysieren, auch wenn es nur kleine Änderungen zum vorherigen Analysedurchgang gab. Diese Eigenschaft machen sich die Autoren zu nutzen, um nur Analyseinformationen an den Stellen neu zu berechnen, die sich geändert haben oder davon beeinflusst werden. Dadurch haben sie Geschwindigkeitsvorteile von bis zu 80% erreicht.

4 Implementierung

In diesem Kapitel wird auf die bei dieser Arbeit implementierten Änderungen an der libFirm Programmbibliothek eingegangen. Im nächsten Abschnitt wird kurz der Zustand beschrieben, wie Datenflussanalysen in libFirm bisher implementiert wurden. Danach wird das implementierte Framework, welches Datenflussanalysen benutzen sollen, vorgestellt. Im folgenden Abschnitt 4.3 und Abschnitt 4.4 werden die Besonderheiten der Iterationsstrategien beziehungsweise der Analysen beschrieben, die speziell bei der Implementierung in libFirm zu beachten waren. Im abschließenden Abschnitt 4.5 wird gezeigt, wie interprozedurale Analyen mithilfe des Frameworks realisiert wurden.

4.1 Bisheriger Zustand

Jede Datenflussanalyse implementierte ihren eigenen Iterationsalgorithmus, meist eine einfache Workliste, bei der ein Knoten auch mehrfach in der Warteschlange vorkommen kann. Der Code ist zum Teil kommentiert, dass es einen besseren Iterationsalgorithmus gibt, der stattdessen implementiert werden sollte. Vermutlich wegen der Einfachheit des Worklist-Algorithmus und Zeitmangels wurde aber kein anderer Algorithmus implementiert oder getestet.

Das Datenflussanalyseframework, das während dieser Arbeit implementiert wurde, behebt diesen Zustand, indem es ein einheitliches Interface für Datenflussanalysen bietet. Die Datenflussanalysen können nun mithilfe des Frameworks auf verschiedene Iterationsalgorithmen zurückgreifen. Dies spart insbesondere bei den komplexeren Algorithmen Implementierungsaufwand.

4.2 Framework in libFirm

Da die Aufgaben für die meisten Analysen gleich sind, wurde ein Framework implementiert um den Iterationsalgorithmus von der Analyse getrennt zu halten und so die Iterationsstrategien leichter austauschen und vergleichen zu können. Auch soll das Framework Implementierungsarbeit sparen, wenn neue Analysen implementiert werden sollen.

Als Erstes wurde ein Framework für Datenflussanalysen als Teil von libFirm erstellt, anfangs nur basierend auf dem Worklist-Algorithmus. Dieses Framework arbeitet auf den Graphen und Knoten dieser Bibliothek und muss daher auch deren Besonderheiten beachten. Es gibt z.B. Knoten mit dem sogenannten Tupel-Modus. Diese allein tragen nichts zu den Datenflussanalysen bei, sondern sind eher organisatorischer Natur und müssen daher bei der Analyse entsprechend übersprungen werden. Auf diesem Framework aufbauend wurden dann mehrere Analysen implementiert beziehungsweise angepasst. Anschließend wurden weitere Iterationsstrategien hinzugefügt und verglichen.

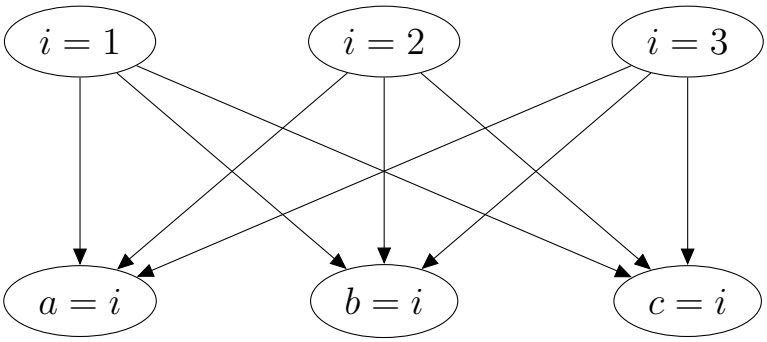
4.2.1 Datenflussanalyse in libFirm

Um den abstrakten Wert einer Variablen an einer bestimmten Programmstelle zu erhalten, müssen die abstrakten Werte aller möglichen Definition der Variablen, die die Programmstelle erreichen können, betrachtet werden. Im Allgemeinen gibt es bei N Definitionen und M Verwendungen $N * M$ use-def Verbindungen. Bei der SSA-Form werden diese durch die ϕ-Funktion minimiert zu $N + M$ Verbindungen und jede Verwendung hat genau eine Definition. Da durch die SSA-Form use-def-Beziehungen bekannt sind, muss nicht mehr für jede Stelle im Programm der komplette abstrakte Programmzustand, also der Zustand aller Variablen, gespeichert werden. Stattdessen kann pro Variable ein abstrakter Datenflusswert eindeutig gespeichert und über die use-def-Beziehungen direkt auf die benötigten Werte zugegriffen werden. Dadurch können Datenflussanalysen auf Programmen in SSA-Form effizienter implementiert werden. Da die SSA-Form in libFirm als Graph implementiert ist, werden außerdem Variablen, die nicht verwendet werden, automatisch aus dem Programm entfernt, da keine Kante mehr zu diesen Knoten führt.

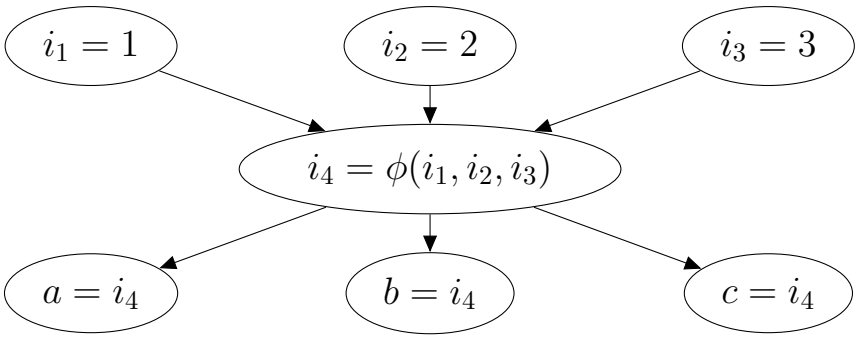
Dadurch, dass Datenflussvariablen nicht pro Block, sondern pro Operation angegeben werden, kann auch die Datenflussanalyse auf den einzelnen Anweisungen anstelle auf Grundblöcken arbeiten. Die Beschränkung auf Grundblöcke war ur-

```
switch (x)
{
case A: i = 1;
case B: i = 2;
case C: i = 3;
};
switch (y)
{
case A: a = i;
case B: b = i;
case C: c = i;
};
```

(a) Code



(b) Use-Def Programm



(c) Use-Def SSA-Form

Abbildung 4.1: Use-Def Beziehungen

```
1 int a = 5;
2 int b = 10;
3 int c;
4 if (x)
5   c = a + b;
6 else
7   c = a + b;
8 return c;
```

(a) Code

```
1 int a1 = 5;
2 int b2 = 10;
3 int c1,c2;
4 if (x)
5   c1 = a1 + b1;
6 else
7   c2 = a1 + b1;
8 int c3 = φ(c1,c2)
9 return c3;
```

(b) SSA-Form

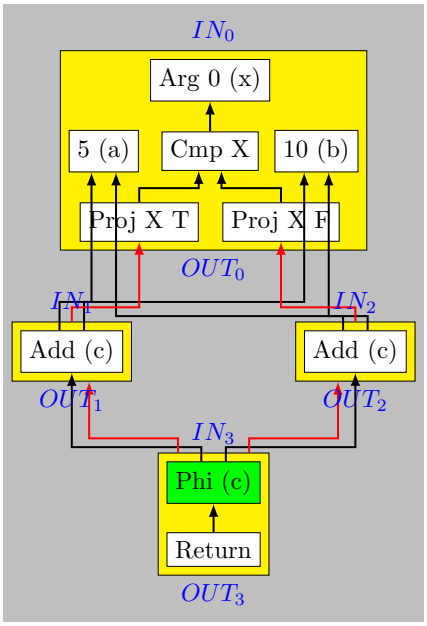
Abbildung 4.2: Beispielprogramm

sprünglich eine Überlegung, Zeit und Platz zu sparen, da in einem Grundblock der Steuerfluss sehr einfach gehalten ist – von Anfang bis Ende ohne Verzweigung. In den meisten Fällen werden aber an einer Stelle nicht die Werte von allen Datenflussvariablen benötigt, sondern nur eine kleine Teilmenge davon. So müssen nur Operationen und deren Datenflusswerte neuberechnet werden, deren Vorgänger sich geändert haben, anstelle aller Operationen eines Grundblockes im Fall, dass sich nur ein von diesem Grundblock verwendeter Datenflusswert geändert hat.

So wird z.B. durch Nutzen der SSA Eigenschaft aus der Conditional Konstantenpropagation die Sparse Conditional Konstantenprogation. Für die Datenflussanalysen in dieser Arbeit stellen die Elemente des Verbandes nicht den gesamten Programmzustand dar, sondern nur den Wert den eine Operation an einer bestimmten Programmstelle hat.

Abbildung 4.2 zeigt ein Beispielprogramm für die Analyse der Konstantenpropagation. Wenn dieses Programmstück untersucht werden soll, ergeben sich 4 Grundblöcke: $(1) - (4)$ ist Grundblock A, (5) Grundblock B, (7) ergibt Grundblock C und (8) ergibt Grundblock D. Es gibt 4 Variablen a, b, c, x. Die Analyse, die auf Grundblöcken arbeitet, benötigt für die Datenflusswerte Platz für # Variablen * 2 * # Grundblöcke Werte, hier also 32. Abbildung 4.3 zeigt die benötigten Datenflusswerte für den Fall, dass pro Grundblock analysiert wird.

Im Gegensatz dazu speichert die auf SSA basierende Analyse für jeden Befehl nur einen Ein- und Ausgangswert. Im Beispiel wird also nur Platz für 14 Werte benötigt (jeder Knoten benötigt 2 Werte). Die für die SSA-Form benötigten Datenflussvariablen werden in Abbildung 4.4 gezeigt.

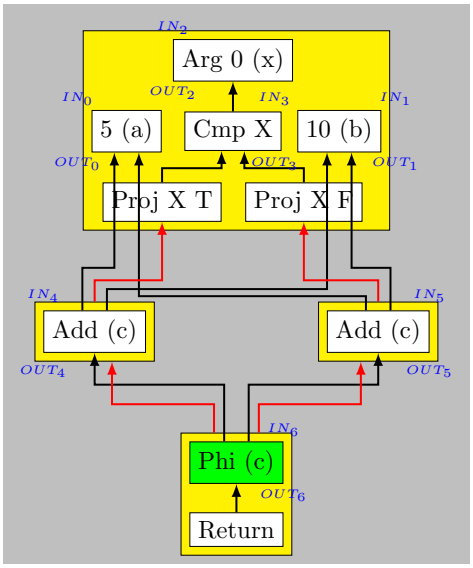


(a) SSA-Graph in libFirm

IN_0	$\{(a,?),(b,?),(c,?),(x,?)\}$
OUT_0	$\{(a,5),(b,10),(c,?),(x,?)\}$
IN_1	$\{(a,5),(b,10),(c,?),(x,?)\}$
OUT_1	$\{(a,5),(b,10),(c,15),(x,?)\}$
IN_2	$\{(a,5),(b,10),(c,?),(x,?)\}$
OUT_2	$\{(a,5),(b,10),(c,15),(x,?)\}$
IN_3	$\{(a,5),(b,10),(c,15),(x,?)\}$
OUT_3	$\{(a,5),(b,10),(c,15),(x,?)\}$

(b) Datenflussvariablen und -werte

Abbildung 4.3: SSA Graph mit Datenflussvariablen pro Block



(a) SSA-Graph in libFirm

IN_0	?
OUT_0	5
IN_1	?
OUT_1	10
IN_2	?
OUT_2	?
IN_3	?
OUT_3	?
IN_4	(5,10)
OUT_4	15
IN_5	(5,10)
OUT_5	15
IN_6	(15,15)
OUT_6	15

(b) Datenflussvariablen und -werte

Abbildung 4.4: SSA Graph mit Datenflussvariablen pro Knoten

4.2.2 Schnittstelle

Das Framework kann für eine Datenflussanalyse mit folgenden Werten parametrisiert werden:

- Richtung: Vorwärts/Rückwarts
- Verbandselemente der Datenflusswerte
- Meet- und Transferfunktion für jeden Operatortyp
- Initialisierung der Datenflusswerte

Damit das Framework mit den Verbandselementen umgehen kann, muss der Benutzer des Frameworks folgende Funktionen angeben:

Alloc	Alloziiere Speicherplatz für ein Verbandselement und setze Wert $\bot$
Copy	Kopiere den Wert
Equal	Überprüfe, ob der Wert gleich ist
Top	Setze den Wert auf $\top$

Bei beiden Richtungen werden die OUT-Werte an den Knoten gespeichert. Bei der Vorwärtsanalyse hat jeder Knoten, da auf SSA gearbeitet wird, pro Operand einen IN-Wert mit Ausnahme des Phi-Knotens. Dieser hat die spezielle Aufgabe die Werte aus mehreren Kontrollflüssen zu vereinen. Hier wird die Meet-Funktion verwendet. Daher kann für jeden Knoten als IN-Wert direkt der OUT-Wert des Vorgängerknotens verwendet werden. Die OUT-Werte werden direkt am Knoten gespeichert.

Bei der Rückwärtsanalyse ist durch die mögliche Verwendung eines Knoten an mehreren Stellen nicht mehr sichergestellt, dass ein Knoten exakt einen IN-Wert hat. Hier muss für jeden Knoten der IN-Wert zusätzlich zum OUT-Wert pro Knoten gespeichert werden, da dieser das Ergebnis der Meet-Funktion aller OUT-Werte der Vorgänger speichert. Dies ist eine Optimierungen, damit die Transferfunktion des Knotens nur ausgeführt werden muss, wenn sich der IN-Wert, also das Meet der Vorgänger OUT-Werte, ändert. Wenn sich ein OUT-Wert eines Vorgängers ändert, jedoch der Wert des Meets gleich bleibt, muss die Transferfunktion nicht erneut ausgeführt werden.

Da in libFirm für einige Analysen nur eine Teilmenge der Knoten des Steuerflussgraphen interessant ist, wird dem Verwender des Frameworks eine Möglichkeit geboten, die verwendeten Knoten einzuschränken. Dadurch können z.B. Zyklen

```
if (A)
        while (B1) B2;
else
        while (C1) C2;
D;
```

(a) Code

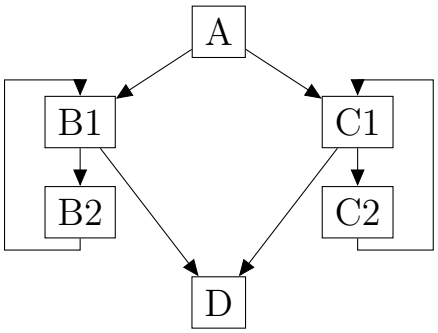


Abbildung 4.5: Beispielgraph

unterbrochen werden, die sonst auf dem ganzen Graphen existieren. Somit kann die Analyse effizienter durchgeführt werden. Dies wird z.B. bei der Don't care Bits Analyse verwendet, um Gleitkomma-Operationen nicht zu analysieren.

Das Ergebnis einer Analyse wird in einem assoziativen Container (`ir_nodemap`) gespeichert, sodass das Ergebnis von einer nachfolgenden Optimierungsphase oder weiteren Analysen verwendet werden kann.

4.3 Iterationsstrategien

In diesem Abschnitt werden die implementierten Iterationsstrategien vorgestellt. Als erstes wird der klassische Algorithmus der Datenflussanalyse vorgestellt. Danach wird der Worklist Algorithmus und seine verschiedenen Implementierungen besprochen. Als letzter Algorithmus wird der Worklist-Algorithmus um starke Zusammenhangskomponenten erweitert. Die hier vorgestellten Iterationsstrategien wurden von Nielson et al. in [27] ausführlich theoretisch behandelt und deren Korrektheit bewiesen. Im folgenden werden sie hier noch einmal kurz vorgestellt.

Abbildung 4.5 zeigt als Beispiel einen Auschnitt eines möglichen Ablaufgraphs mit dessen Hilfe die Iterationsreihenfolgen der in diesem Abschnitt vorgestellten Algorithmen erläutert wird.

4.3.1 Round-Robin

Der grundlegende Algorithmus um die Datenflussgleichungen zu lösen ist der *round-robin* iterative Algorithmus, dargestellt in Listing 4.1.

Listing 4.1: Round-Robin-Algorithmus

```
// Initialisierung
for all x ∈ X do
        Analysis[x] := ⊥;
changed := true

// Iterieren
while changed do
        changed := false
        for i := 1 to N do
                new := eval(t_i, Analysis);
                if new ⋢ Analysis[x_i] then
                        changed := true
                        Analysis[x_i] := Analysis[x_i] ⊓ new
```

Als Reihenfolge der Knoten in einer Runde wurde die Reverse Postorder des Graphen festgelegt. Diese ist optimal, wenn der Graph keine Zyklen enthält. Anschaulich kann man sich vorstellen, dass ein Knoten erst ausgeführt wird, nachdem seine Vorgänger ausgeführt wurden. Dieser Algorithmus wird im folgenden RR genannt. Eine zweite Variante dieses Algorithmus sortiert die Knoten erst in Reverse Postorder der Blöcke und innerhalb eines Blockes noch einmal in Reverse Postorder. Diese Variante nennen wir RR-B.

Eine mögliche RPO-Reihenfolge für den Beispielgraph ist A, C1, C2, B1, D, B2. Wie man sieht, kommt B2 erst am Ende, obwohl D am Ende des Programmflusses ist. Dies führt dazu, dass eine Änderung in B2 mindestens 2 Iterationen über alle Knoten des Graphens braucht, um zu D zu gelangen. Änderungen von B2 können z.B. die Knoten A, C1 und C2 nicht betreffen, jedoch müssen diese von dem Algorithmus jedes Mal betrachtet werden.

4.3.2 Worklist

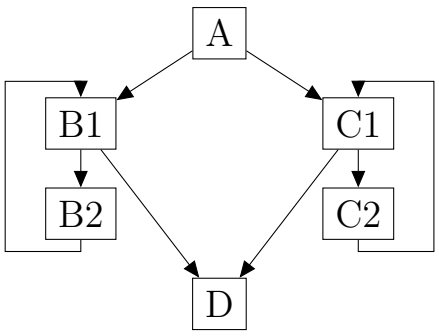
Wie wir gesehen haben, werden beim Round-Robin Algorithmus oft Knoten betrachtet, die sich nicht ändern, weil bei jeder Iteration der gesamte Programmgraph iteriert wird. Der Worklist Algorithmus wurde dafür entwickelt, diese zusätzliche Arbeit zu vermeiden und nur die Knoten zu betrachten, bei denen es Änderungen geben kann. Der abstrakte Worklist Algorithmus wird in Listing 4.2 dargestellt. Wie schon in Kapitel 3 vorgestellt, haben Cooper et al. diesen Algorithmus mit verschiedenen Containern als Workliste evaluiert. Wir wiederholen

dies hier für einige Implementierungen um die Ergebnisse zu vergleichen. In dieser Arbeit wurde eine Warteschlange verwendet. Diesen Algorithmus nennen wir im folgenden QUEUE.

Listing 4.2: Abstrakter Worklist-Algorithmus

```
// Initialisierung
W := empty;
for all x in S do
        W := insert(x, W);
        Analysis[x] := ⊥;

// Iterieren
while W ≠ empty do
        (x, W) = extract(W);
        new := transfer(x);
        if new ⋢ Analysis[x] then
                Analysis[x] := Analysis[x] ⊓ new;
                for all y in pred(x) do
                        W := insert(y, W);
```



(a) Beispielgraph

Aktuell	Worklist	neu hinzugefügt
A		B1, C1
B1	C1	B2, D
C1	B2, D	D, C2
B2	D, D, C2	B1
D	D, C2, B1	
D	C2, B1	
C2	B1	C1
B1	C1	
C1		

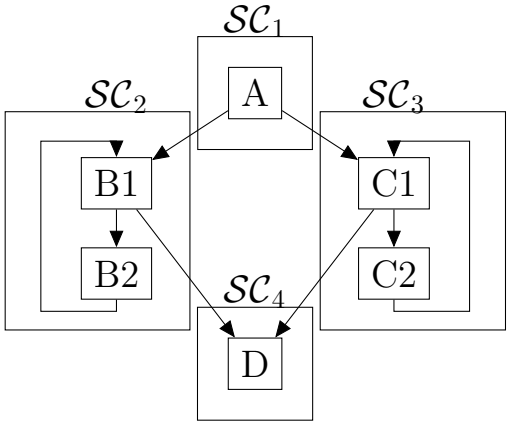
(b) Abarbeitungsreihenfolge

Abbildung 4.6: Beispiel Worklist-Algorithmus mit Queue

Als weitere Implementierung wurde eine Prioriätsqueue verwendet. Bei diesem Algorithmus wird zuerst die reverse Postorder der Knoten berechnet und diese Reihenfolge wird als Priorität verwendet. Diesen Algorithmus nennen wir RPO. Eine Variante davon berechnet die reverse Postorder der Grundblöcke, sowie die reverse Postorder der Knoten innerhalb des Grundblockes und kombiniert diese Reihenfolgen. Erst alle Knoten eines Grundblockes in reverse Postorder, danach nach reverse Postorder der nächste Grundblock und dessen Knoten. Diese Variante wird B-RPO genannt.

4.3.3 Starke Zusammenhangskomponente (Scc)

Eine Technik, die gute Performanz verspricht, ist das Identifizieren und Verarbeiten von starken Zusammenhangskomponenten. Der Idee dabei ist, dass die Zusammenhangskomponenten einen azyklischen Graphen ergeben und sich Änderungen in einer Zusammenhangskomponente nicht auf Knoten einer vorherigen Zusammenhangskomponenten auswirken können. Die Zusammenhangskomponenten werden in topologischer Reihenfolge besucht, SC_1 vor SC_2 falls es eine Kante von SC_1 zu SC_2 gibt. Diese Reihenfolge wird mithilfe umgekehrte post-order auf dem reduzierten Graph der Zusammenhangskomponenten berechnet. Die Knoten innerhalb einer Zusammenhangskomponente werden in mithilfe einer Prioritätsqueue mit umgekehrter post-order solange iteriert, bis sich die Werte aller Knoten der Zusammenhangskomponente stabilisiert haben. So wird die Anzahl der Transferfunktionsaufrufe gegenüber der Reihenfolge, die durch den Algorithmus zur SCC Berechnung entsteht, weiter verringert, wobei der Zeitaufwand nur geringfügig höher ist. Erst dann werden Knoten der nächsten Zusammenhangskomponente besucht. Dieser Algorithmus nennen wir Scc.



(a) Beispielgraph

Aktuell	Worklist	neu hinzugefügt
A		$\mathcal{SC}_2$,$\mathcal{SC}_3$
B1	$\mathcal{SC}_3$	B2, $\mathcal{SC}_4$
B2	$\mathcal{SC}_3$,$\mathcal{SC}_4$	B1
B1	$\mathcal{SC}_3$,$\mathcal{SC}_4$	
C1	$\mathcal{SC}_4$	C2
C2	$\mathcal{SC}_4$	C1
C1	$\mathcal{SC}_4$	
D		

(b) Abarbeitungsreihenfolge

Abbildung 4.7: Beispiel Scc-Algorithmus

Bei Beispiel 4.7 ergeben sich die zwei möglichen Reihenfolgen der Zusammenhangskomponenten: {A}, {B1, B2}, {C1, C2}, {D} und {A}, {C1, C2}, {B1, B2}, {D}, wobei hier nur die erste Reihenfolge betrachtet wird. Der Algorithmus wird nun erst den Knoten A betrachten und dann den Knoten B1. Knoten B1 und B2 werden so lange iteriert, bis sich ein lokaler Fixpunkt eingestellt hat. Erst dann wird zur nächsten SCC mit den Knoten C1 und C2 übergegangen, die ebenfalls bis zum lokalen Fixpunkt iteriert werden. Ganz am Ende wird dann Knoten D iteriert und danach der Algorithmus beendet.

Da dieser Algorithmus die Zusammenhangskomponenten berechnen muss, hat er einen großen initialen Aufwand. Es wurde erwartet, dass diese Iterationsstrategie

die Anzahl der ausgeführten Transferfunktionen minimiert. Dadurch könnte dem höheren Aufwand beim Initialisieren entgegengewirkt werden, so dass der Gesamtaufwand geringer als bei den anderen Algorithmen sei. Die Ergebnisse der Experimente zeigen jedoch, dass dies erst bei sehr großen Graphen der Fall ist. Bei den in dieser Arbeit betrachteten intraprozeduralen Analysen ist die Anzahl der Knoten jedoch zu gering.

4.4 Analysen

Um das Framework zu testen wurde als erstes die Value Range Propagation implementiert, da es diese Optimierung schon in libFirm gibt. Die Analysen werden in der Reihenfolge aufgelistet, in der sie auch im cparser ausgeführt werden.

Allen hier implementierten Analysen ist gleich, dass sie keine Werte im Speicher modellieren, also alle libFirm Knoten des Typs *memory* mithilfe von Filtern ignorieren. Für die theoretische Betrachtung der Datenflussanalysen sei auf die Grundlagen in Abschnitt 2.4 verwiesen.

4.4.1 Postdominanz

Der Algorithmus zu Berechnung der Postdominanz muss nur die Grundblöcke als Knoten des Graphen betrachten. Da in Firm die Grundblöcke nicht direkt miteinander verbunden sind, sondern nur indirekt über Steuerflussknoten, müssen diese bei der Postdominanzberechnung berücksichtigt werden. Für die Analyse wird ein Filter vorgegeben, der nur Knoten des Typs Block und Steuerflussknoten zulässt. Der Verband besteht aus der Potenzmenge $P(V)$ der Menge aller Knoten des Graphen mit der Vereinigung, also $(P(V), \subseteq)$. Er wird mithilfe eines Bitvektors implementiert. Für Blöcke berechnet die Transferfunktion $transfer_{block}$ den Wert nach den Datenflussgleichungen in Unterabschnitt 2.4.1. Für die Steuerflussknoten werden als Transferfunktion $transfer_{cfn}$ die Verbandswerte ihres nachfolgenden Blockes einfach weitergegeben.

$$transfer_{block}(N) = (\bigcap_{i \in pred(N)} \mathrm{IN}_i(N)) \bigcup \{N\}$$
$$transfer_{cfn}(N) = \mathrm{IN}_0(N)$$

In libFirm war die Berechnung der Postdominanz schon mit Hilfe des Lengauer-Tarjan Algorithmus [24] implementiert. Dieser berechnet die direkten Postdominatoren anstelle der Menge aller Postdominatoren und wird von libFirm an verschiedenen Stellen verwendet. Daher wurde diese Analyse vorerst nur zu Messzwecken in dieser Arbeit entwickelt.

4.4.2 Konstantenpropagation

Für die Konstantenpropagation wird ein Verband verwendet, der aus allen möglichen Variablenwerten besteht, sowie den zwei speziellen Werten $\bot$ und $\top$ – wie schon in im Beispiel in Abbildung 2.9 für Integervariablen gezeigt.

Eine anschließende Optimierungsphase ersetzt alle Knoten, die nicht vom Typ Konstante sind, aber durch die Analyse als konstant erkannt worden sind, durch einen neuen Konstanten-Knoten.

4.4.3 Value Range Propagation

In der bisherigen Implementierung in libFirm werden die Wertebereiche aller Integervariablen mithilfe von gesetzten und nicht gesetzten Bits dargestellt. Diese Darstellung ist relativ ungenau, hat jedoch den Vorteil, dass der Speicherbedarf im Gegensatz zu z.B. der Methode von Patterson sehr gering ist. Diese bisherige Implementierung wurde für diese Arbeit so angepasst, dass das Framework verwendet wird anstelle des einfachen Worklist-Algorithmus. Dadurch konnte der unterliegende Iterationsalgorithmus einfach gewechselt werden, ohne weitere Änderungen an der ursprünglichen Analyse vornehmen zu müssen.

Die Analyse berechnet auf Integerwerten, welche Bits gesetzt, nicht gesetzt, unbekannt oder nicht initialisiert sind. In der nachfolgenden Optimierungsphase können anhand der Analyseergebnisse Knoten durch Konstanten ersetzt oder durch weniger aufwendige Operationen ersetzt werden.

4.4.4 Don't-care Bits

Die bisher in libFirm vorhandene Don't-care-bits Analyse von Seltenreich [30], wurde für diese Arbeit an das Framework angepasst. Da ein paar Veränderungen

an der Transferfunktion notwendig waren, wurden in einem ersten Schritt die Originalanalyse und die neu angepasste Analyse zusammen ausgeführt und deren Ergebnisse verglichen. Dies stellte sicher, dass sich bei der Umwandlung keine Fehler einschlichen und die Analyse immer noch korrekt ist. Im nachfolgenden Schritt wurde die Originalanalyse entfernt und komplett durch die angepasste Analyse ersetzt.

Die implementierte Analyse verwendet als Verband für Integerwerte einen Bitvektor der jeweiligen Länge des Wertes. Für jedes Bit sind die Zustände $\bot = 0$ und $\top = 1$ möglich, die darstellen, ob dieses Bit verwendet wird ($\top$)oder nicht ($\bot$).

4.5 Interprozedurale Analyse

In einem weiteren Schritt wurde das Framework um die Möglichkeit erweitert, eine Analyse interprozedural durchzuführen. Da hierfür nur wenig Zeit eingeplant war, aber auch Kontextsensitivität erwünscht war, wurde ein Algorithmus implementiert, der an die Stellen der Funktionsaufrufe den Funktionsrumpf kopiert. Da ein komplettes Kopieren des Graphen sehr aufwändig ist, werden nur die Daten, die für den Kontext wichtig sind, für die Analyse kopiert, z.B. muss der Funktionsgraph selbst nicht kopiert werden, da dieser von jeder Aufrufstelle aus gleich ist. Die benötigen Informationen beschränken sich auf die abstrakten Werte der einzelnen Knoten. Für jeden Prozeduraufruf wird ein eigener sogenannter Kontext erzeugt, der Parameter und Ergebnis, sowie den Zustand der einzelnen Knoten beinhaltet. Da bei rekursiven Funktionen die Anzahl der Kontexte potentiell unendlich ist, aber nur endlich viel Speicher zur Verfügung steht, wurden zwei Einschränkungen implementiert.

Der Benutzer des Frameworks gibt eine maximale Tiefe an, bis zu der Prozeduraufrufe verfolgt werden sollen. Damit kann auch die interprozedurale Analyse ausgeschaltet werden, falls feststeht, dass sie für eine Analyse keinen Informationsgewinn bringt. Bei Erreichen der festgelegten Tiefe oder für den Fall, dass libFirm zu einer Prozedur kein Graph vorliegt, der analysiert werden kann, werden für die OUT-Werte $\top$ vergeben um so ein gültiges, wenn auch möglicherweise suboptimales, Ergebnis zu erzielen. Die zweite Einschränkung ist die maximale Anzahl von Prozeduraufrufen, die analysiert werden.

Die Verbandswerte der Variablen, mit denen eine Prozedur aufgerufen wird, werden durch das Framework von der Aufrufstelle in den Graphen der Prozedur

an die Stellen der Prozedurparameter kopiert und danach der Graph der Prozedur analysiert. Wenn sich ein Fixpunkt auf dieser Prozedur eingestellt hat, wird das Ergebnis wieder zurück an die Stelle des Prozeduraufrufes im vorherigen Graphen kopiert und dort die Analyse fortgesetzt. Wenn sich der Wert eines Parameters erneut ändert, werden im Graph der aufgerufenen Prozedur nur die sich ändernden Datenflusswerte erneut berechnet.

Einige Iterationsstrategien berechnen vor der eigentlich Fixpunktberechnung verschiedene Daten wie eine Reihenfolge aller Knoten. Diese Berechnungen sind meist sehr zeitaufwändig, aber sind für den betrachteten Prozedurgraphen konstant. Aus diesem Grund werden diese Ergebnisse pro Prozedur gespeichert und werden bei jedem Kontext dieser Prozedur wiederverwendet. Davon profitiert im Besonderen der Scc Algorithmus.

In libFirm wird eine Optimierung und daher auch die Analyse auf allen Prozeduren ausgeführt, bevor die nächste Optimierung angewandt wird. Dadurch wird für die interprozedurale Analyse immer jeweils eine Prozedur als Startpunkt verwendet, von der der Aufrufgraph analysiert wird. Da nach der Analyse einer Prozedur normalerweise direkt eine Optimierungsphase folgt, bei der der Inhalt der Prozedur geändert wird, muss jede Analyse einer anderen Prozedur alle Informationen neu berechnen. So werden für das Beispiel in Abbildung 4.8 die Prozeduren `test1` und dann `test2` analysiert und optimiert. Die interprozedurale Analyse wird dabei die Prozedur `calc` aus dem Kontext in `test1` und in `test2` betrachten, muss jedoch alle zusätzlichen Informationen wie die Reihenfolge der Knoten dabei neuberechnen, da zwischen dem Analysieren der beiden Kontexte eine Optimierungsphase liegt, die Änderungen am Programm vornehmen könnte. Wenn Prozedur `test3` analysiert wird, können diese zusätzlichen Informationen für Prozedur `calc` allerdings sowohl für Kontext `test1` als auch `test2` verwendet werden ohne neuberechnet zu werden, da jetzt keine Optimierungsphase dazwischen liegt.

Ein anderer möglicher Ansatz wäre, einen kompletten Aufrufgraph aufzubauen und diesen zu analysieren. Dies verspricht weniger benötigte Analysen und dadurch eine schnellere Bearbeitung, jedoch auch viele umfangreiche Änderungen an der Art und Weise wie Optimierungen in libFirm ausgeführt werden und wurde daher nicht im Rahmen dieser Arbeit implementiert. Ein weiterer Vorteil wäre, dass für Parameter von Prozeduren, die nur in dem analysierten Programm verfügbar sind, die Menge der möglichen abstrakten Werte anhand der Kontexte der aufrufenden Prozeduren eingeschränkt werden kann. Somit könnten für die Prozedurparameter statt dem Wert $\top$, also keine Information, das Meet aus allen Variablen der Prozeduraufrufe verwendet werden, welches potentiell mehr Informationsgehalt bestitzt, und damit bessere Ergebnisse erreicht werden. In dem in

```
int calc(int a, int b)
{
        if (a>10)
                return a - 10;
        return a + b;
}

void test1()
{
        ...
        t2 = calc(2, t1);
        ...
}

void test2()
{
        ...
        t4 = calc(5, t3);
        ...
}
void test3()
{
        ...
        test1();
        ...
        test2();
        ...
}
```

Abbildung 4.8: Programmbeispiel

Abbildung 4.8 gezeigten Beispiel könnte die Prozedur `calc` vereinfacht werden, da alle Aufrufe für den Parameter a einen Wert kleiner 10 haben und daher die `if` Bedingung immer fehlschlägt und entfernt werden kann.

5 Evaluation

In diesem Kapitel werden die Ergebnisse unserer Untersuchung vorgestellt. Als erstes werden der Testaufbau und die Messwerte gezeigt. Danach werden die einige Iterationsstrategien miteinander verglichen.

5.1 Testaufbau

C Programme werden vom CPARSER compiliert und anschließend die Informationen des hinzugefügten Frameworks ausgewertet. Aus dem SPEC CINT2000 Benchmark [14] wurden die C Programme analysiert. Als Testrechner wurde ein Intel Pentium Dual-Core E5300 mit 2.6 GHz und 4GB RAM mit Ubuntu und Linux Kernel Version 3.8.0 verwendet.

Um die Performanz einer Iterationsstrategie zu messen, wurden zwei Metriken verwendet. Die eine ist die Anzahl wie häufig die Transferfunktion der Knoten des Graphes ausgeführt wird. Die andere ist die Zeit, die für die Analyse benötigt wurde. Alle in dieser Arbeit verwendeten Analysen außer der Postdominanz, analysieren als ersten Phase von Optimierungen den Graph der Funktion mithilfe des Frameworks analysiert. Im nächsten Schritt wird das Ergebnis der Analyse dann auf den Prozedurgraph angewandt. Gemessen wird hierbei nur der Analyseteil.

Für die interprozeduralen Analysen wurde die maximale Tiefe auf 2 Prozedurkontexte limitiert, da sonst der benötigte Speicher das Maximum des Testrechners überschritten hätte.

5.2 Messwerte und Vergleich

Wie man in Abbildung 5.1 sieht, führt der SCC-Algorithmus die Transferfunktionen am wenigstens häufig aus. Danach folgen die Worklist-Algorithmen, die

Analyse	RR	QUEUE	B-RPO	RPO	SCC
Konstantenpropagation	1 719	698	280	344	275
Dont-care bits	625	325	298	312	276
Value Range Propagation	952	393	166	205	164
Postdominanz	227	288	88	99	86
Durchschnitt	947	448	214	248	206

Abbildung 5.1: Anzahl ausgeführter Transferfunktionen, gemittelt über alle Graphen der SPEC-Programme

Analyse	Zeit (µs)	Zeit (%) *
RR	1 236	-
QUEUE	984	-21
RPO	935	-25
B-RPO	956	-23
SCC	1 488	+20

* relativ zu RR

Abbildung 5.2: Zeiten

nach RPO priorisieren. Der Worklist-Algorithmus QUEUE, der ohne Priorisierung arbeitet, braucht fast doppelt so viele Transferfunktionsberechnungen. Wie zu erwarten hat der Round-robin Algorithmus die höchste Anzahl an Transferfunktionen.

Wenn man jedoch die Zeiten in Abbildung 5.2 betrachtet, sieht man, dass der Aufwand des SCC Algorithmus, eine gute Iterationsreihenfolge zu erreichen, so hoch ist, dass andere Algorithmen mit schlechteren Reihenfolgen schneller sind. Die beiden Worklist-Algorithmen mit RPO-priorisierten Warteschlangen schneiden hierbei am besten ab. Wie schon Cooper et al. gezeigt haben, benötigt der Round-robin Algorithmus am meisten Zeit.

5.2.1 Vergleich Starke Zusammenhangskomponente

Wie wir in Abbildung 5.4 sehen, hat der SCC Algorithmus im Durchschnitt die geringste Anzahl an Transferfunktionsaufrufen. Bei genauerer Analyse sieht man jedoch, dass bei 2,5% der analysierten Prozeduren der B-RPO Algorithmus sogar eine noch geringere Anzahl aufweist. Dies führte zur Frage, in welchen Situationen dies passieren kann. In Abbildung 5.5 wird ein Beispiel skizziert, bei dem der SCC Algorithmus eine schlechtere Iterationsreihenfolge als der B-RPO Algorithmus berechnen kann. Das Problem hierbei sind die Datenabhängigkeiten, die

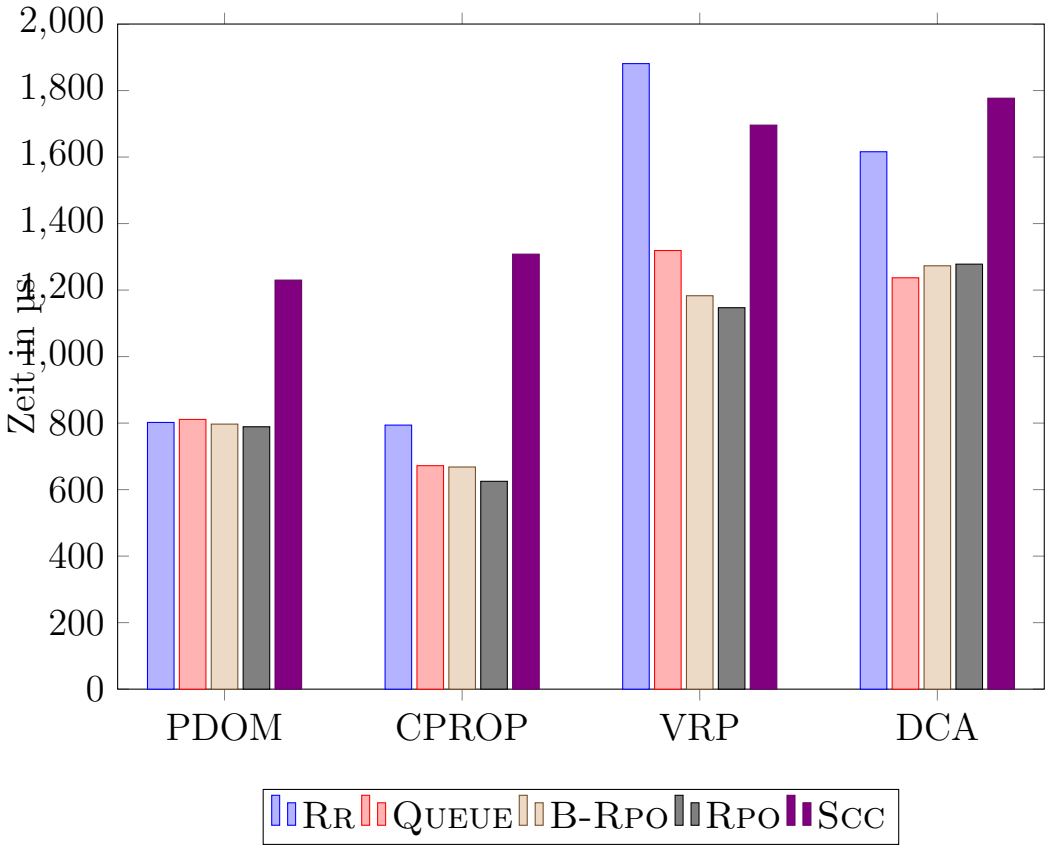


Abbildung 5.3: Zeiten intraprozeduraler Analyse

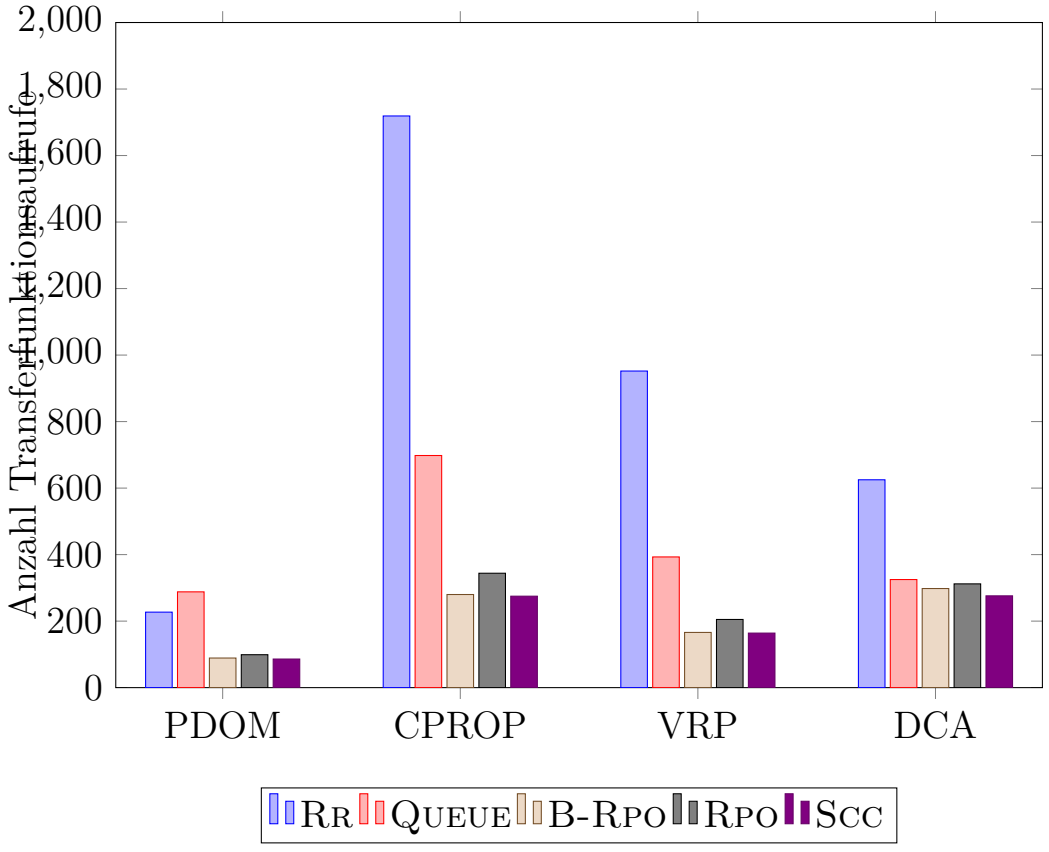


Abbildung 5.4: Transferanzahl intraprozeduraler Analyse

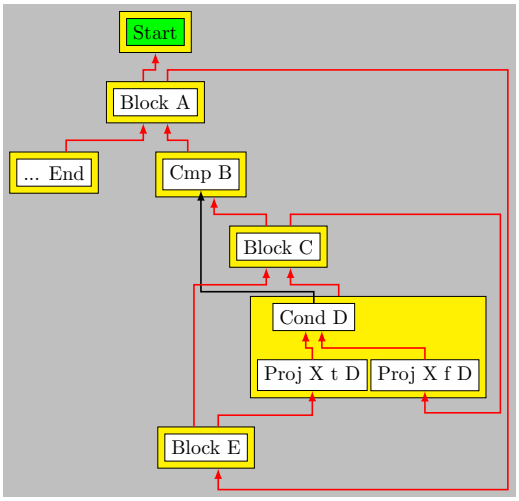


Abbildung 5.5: Beispielgraph SCC vs. BRPO

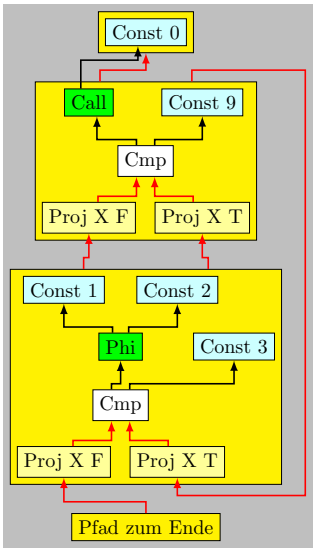


Abbildung 5.6: Beispiel Konstantenpropagation

alle Knoten rechts unterhalb `Block A` einschließlich zu einer Zusammenhangskomponente verbinden. Durch die Datenabhängigkeit `Cond D` → `Cmp B` kann beim Erstellen der RPO innerhalb der SCC der Knoten `Block C` erst übersprungen und durch das Traversieren der Nachfolgeknoten erreicht werden. Dadurch kommt der Knoten `Block C` in der RPO nach dem Knoten `Cond D`, obwohl er im Programmfluss auf jeden Fall vorher ausgeführt wird.

Abbildung 5.6 zeigt ein zweites Beispiel, bei dem die Reihenfolge des Scc Algorithmus schlechter ist als die des B-Rpo Algorithmus. Die Reihenfolge in der die beiden Iterationsstrategien vorgehen, ist in Abbildung 5.7 beschrieben.

Am Beispiel der Vorwärtsanalyse: Die Reihenfolge des SCC-Algorithmus ist nicht gut, da es eine Stelle gibt, in der die Transferfunktion des Knoten vor dessen Block

Scc	BRpo
Block1	Block1
Call	Call
Cmp	Cmp
Proj X F	Proj X F
Block2	Proj X T
Phi	Block2
Cmp	Phi
Proj X F	Cmp
Block1	Proj X F
Proj X T	Block1
Block2	
Phi	
Cmp	
Proj X F	

Abbildung 5.7: Vergleich Scc und BRpo für Beispiel aus Abbildung 5.6

ausgeführt wird. Im allgemeinen Fall wird zwar ein Block mit Vorgängern ausserhalb der SCC bevorzugt, bei Knoten innerhalb der SCC kann jedoch durch die RPO Berechnung auch ein Knoten vor dessen Block priorisiert. Dies führt dazu, dass in einer SCC, die einer Teilmenge der ursprünglichen SCC entspricht, diese Bevorzugung nicht stattfindet und so eine ungünstigere Reihenfolge berechnet wird. Dies könnte durch noch mehr Aufwand beim Berechnen der Iterationsreihenfolge umgangen werden, allerdings ist dieser Aufwand bei der SCC schon so hoch, dass andere Algorithmen zeitlich besser abschneiden. Für den Fall, dass eine Analyse eine so aufwändige Transferfunktion hat, dass dieser Aufand den Großteil der Analyse ausmacht, kann es lohnenswert sein, diese Iterationsstrategie weiter zu analysieren. Da dies bei den hier betrachteten Analysen nicht der Fall ist, wurde sie jedoch nicht weiter vertieft.

5.2.2 Interprozedurale Analyse

Bei der interprozeduralen Analyse gab es Probleme mit dem B-Rpo Algorithmus bei einigen Testprogrammen, die nicht mehr während der festgelegten Bearbeitungszeit behoben werden konnten. Daher fehlen die Ergebnisse für diesen Algorithmus. Es ist jedoch zu erwarten, dass das Ergebnis für den B-Rpo Algorithmus wie im intraprozeduralen Fall ähnlich dem Rpo Algorithmus ist, da die beiden Algorithmen sehr ähnlich sind.

Programm	intraprozedural	interprozedural	Steigerung
164.gzip	0	0	0%
175.vpr	5	5	0%
176.gcc	335	347	3.6%
181.mcf	0	0	0%
186.crafty	4	8	100%
197.parser	15	15	0%
253.perlbmk	48	231	381%
254.gap	40	40	0%
255.vortex	24	24	0%
256.bzip2	0	0	0%
300.twolf	7	7	0%
complete	478	677	41.6%

Abbildung 5.8: Ersetzte Konstanten bei intraprozeduraler und interprozeduraler Konstantenpropagation

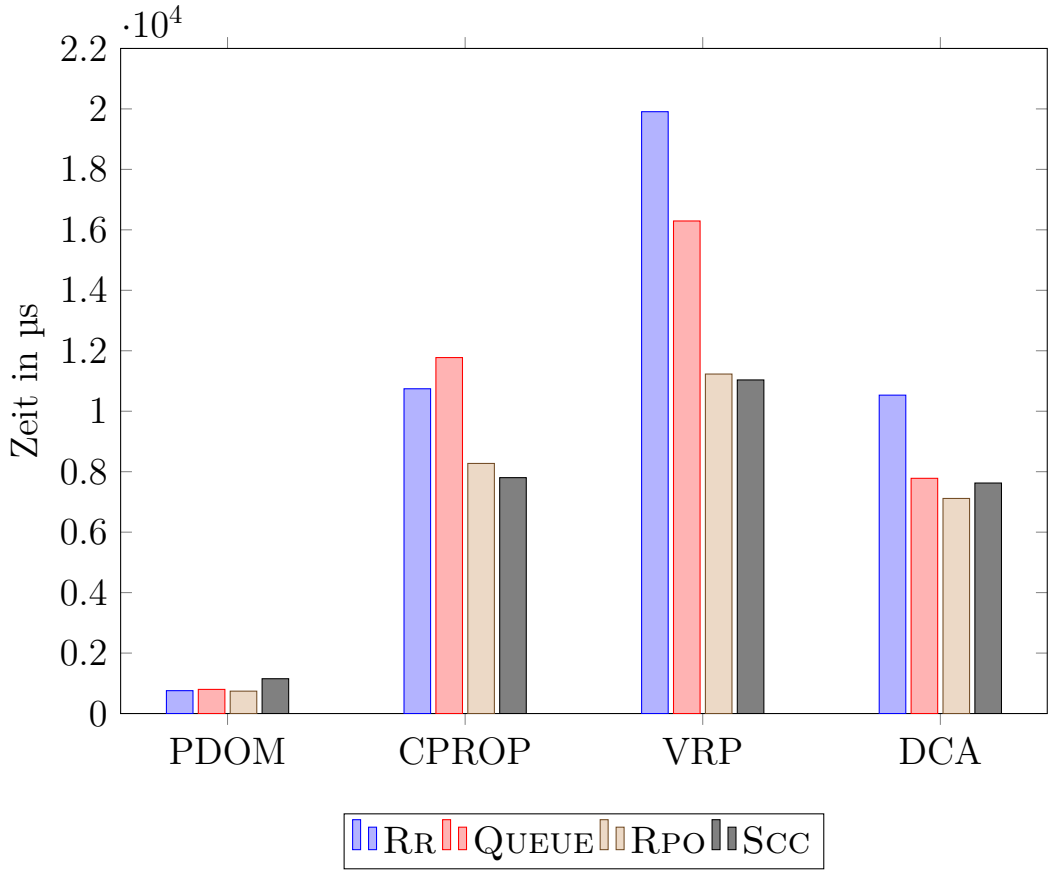


Abbildung 5.9: Zeiten interprozeduraler Analyse

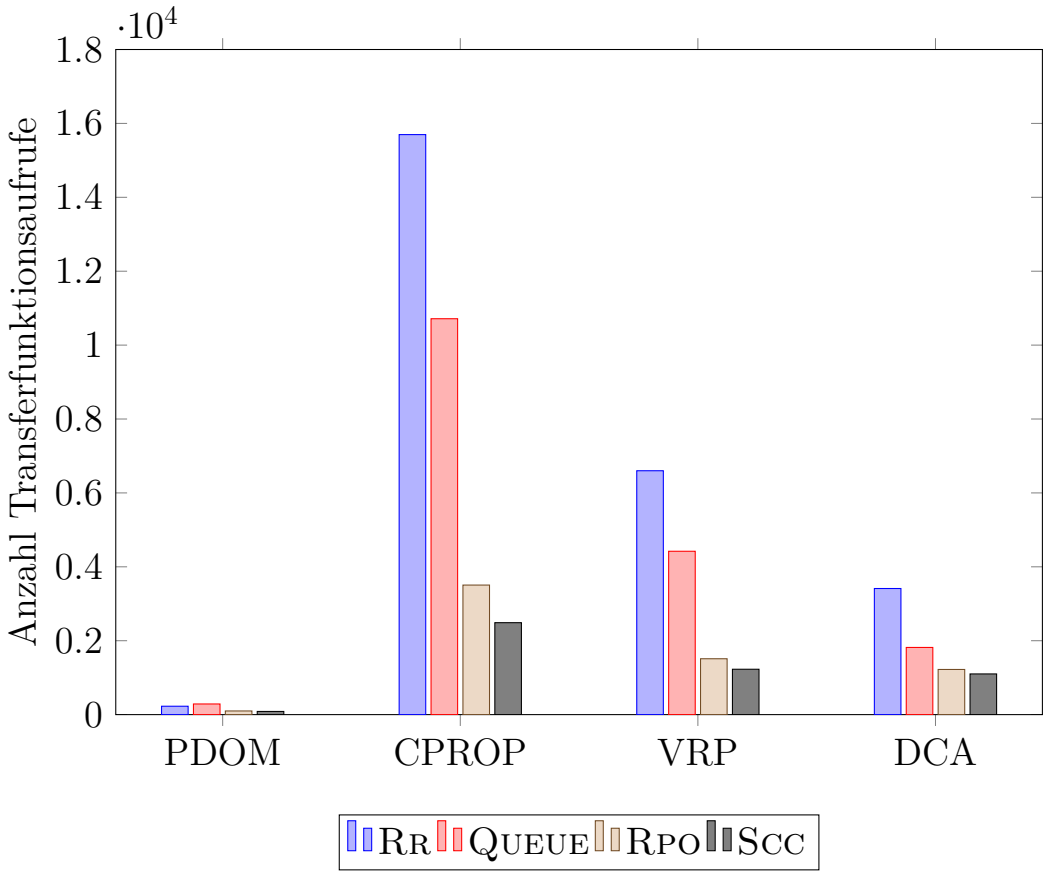


Abbildung 5.10: Transferanzahl interprozeduraler Analyse

Um zu messen, dass die interprozeduralen Datenflussanalysen ein besseres Analyseergebnis gegenüber der intraprozeduralen Analyse berechnen, wurde anhand der Konstantenpropagation die Anzahl der durch Konstanten ersetzten Knoten gemessen. Der gewählte Iterationsalgorithmus spielt dabei keine Rolle. Während bei internen Tests mit interprozeduraler Analyse 7,5 mal so viele Knoten ersetzt wurden wie mit intraprozeduraler Analyse, waren es bei den repräsentativeren SPEC Programmen weitaus weniger. Wie in Abbildung 5.8 zu sehen, ist nur ein Zuwachs bei 3 Programmen zu sehen. Den besten Zuwachs hat dabei *253.perlbmk* mit fast 5 mal so viel ersetzter Konstanten. Im Durchschnitt wurden ungefähr 40% mehr Konstanten ersetzt. Man sieht, dass die Möglichkeit zur Verbesserung der Ergebnisse durch interprozedurale Analyse sehr stark von der Art der Programme abhängig ist.

Das Verhältnis der Anzahl der ausgeführten Transferfunktionen für die verschiedenen Iterationsalgorithmen, zu sehen in Abbildung 5.10 ist ähnlich dem Ergebnis für den intraprozeduralen Fall.

6 Zusammenfassung und Ausblick

Da der Algorithmus mit den Zusamenhangskomponenten einen so hohen Initialaufwand hat, der sich erst für interprozedurale Analysen relativiert, ist in der Praxis der RPO oder B-RPO Algorithmus besser geeignet. Erst bei großen Graphen, die z.B. durch die interprozedurale Analyse implizit entstehen, wird der Aufwand der zusätzlichen Berechnung der starken Zusammenhangskomponenten relativiert, so dass der SCC Algorithmus zusammen mit dem RPO Algorithmus zu den schnellsten der in dieser Arbeit untersuchten Algorithmen gehört. Ein weiterer Vorteil für den RPO Algorithmus ist Einfachkeit, mit der er zu implementieren ist.

Für den Fall, dass bei einer Analyse die Transferfunktion einen größeren Anteil am Gesamtaufwand als bei unseren getesteten Analysen darstellt, ist zu erwarten, dass der SCC Algorithmus besser abschneidet, da er die geringste Anzahl an Transferfunktionsaufrufen hatte.

Mithilfe des Frameworks kann eine intraprozedurale Analyse in eine intraprozedurale Analyse umgewandelt werden. Am Beispiel der Konstantenpropagation haben wir jedoch gesehen, dass ein Großteil der analysierten Programme keinen Vorteil aus der interprozeduralen Analyse zieht. Da der zusätzliche Zeitaufwand bei der interprozeduralen Analyse sehr hoch ist, muss man die Verwendung im Einzelfall abwägen.

6.1 Erweiterungen und Verbesserungen

Da das Thema interprozedurale Analyse ein recht großes Gebiet umfasst und in dieser Arbeit nur am Rande betrachtet wurde, gibt es hier großes Potenzial für interessante Erweiterungen und Optimierungen. Eine Optimierung wäre z.B. aufgerufene Prozeduren einmal für alle Parameter mit dem Wert $\bot$ zu analysieren und falls das Ergebnis $\top$ ergibt, können Prozeduraufrufe an allen Stellen direkt durch Bottom ersetzt werden, da für diese Prozedur unabhängig der Da-

tenflusswerte der Parameter immer der Wert $\top$ als Ergebnis geliefert wird. Auch könnte interessant sein, mehrere Aufrufstellen mit gleichen Parameterwerten zusammenzufassen, um Berechnungsarbeit zu sparen. Es könnten auch Strategien, wann und wo die Aufrufkontexte zu erstellen sind, untersucht werden.

Da der favorisierte Algorithmus mit den Zusamenhangskomponenten nicht die beste Performance bietet, sollten weitere Algorithmen untersucht werden, ob ein besserer gefunden werden kann.

Eine weitere mögliche Erweiterung des Frameworks ist die Unterstützung von der Beschleunigungstechnik Widening und Narrowing, die bei Verbänden großer bzw. unendlicher Höhe verwendet werden kann. Damit kann eine Datenflussanalyse mit einem Verband von unendlicher Höhe auch mit einem iterativen Algorithmus gelöst werden. Unter zu Hilfenahme von Ideen aus [4] können geeignete Stellen für die Anwendung der Widening- und Narrowing-Operatoren gefunden werden.

Die am Ende von Abschnitt 4.5 angedeutete Veränderung des Trennen von Analyse- und Optimierungsphasen verspricht durch Zwischenspeichern von Informationen wie der Iterationsreihenfolge eine Beschleunigung der interprozeduralen Analyse.

Literaturverzeichnis

[1] Adl-Tabatabai, Ali-Reza ; Gross, Thomas ; Lueh, Guei-Yuan: Code reuse in an optimizing compiler. In: *SIGPLAN Not.* 31 (1996), Oktober, Nr. 10, 51–68. http://dx.doi.org/10.1145/236338.236342. – DOI 10.1145/236338.236342. – ISSN 0362–1340

[2] Arzt, Steven ; Bodden, Eric: Efficiently updating IDE-based data-flow analyses in response to incremental program changes / Technische Universität Darmstadt. 2013 (TUD-CS-2013-0253). – Technical Report

[3] Beck: *Test Driven Development: By Example*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321146530

[4] Bourdoncle, François: Efficient chaotic iteration strategies with widenings. In: *Formal Methods in Programming and Their Applications*, Springer-Verlag, 1993, S. 128–141

[5] Braun, Matthias ; Buchwald, Sebastian ; Hack, Sebastian ; Leissa, Roland ; Mallon, Christoph ; Zwinkau, Andreas: Simple and Efficient Construction of Static Single Assignment Form. Version: 2013. http://www.cdl.uni-saarland.de/projects/ssaconstr. In: Jhala, Ranjit (Hrsg.) ; Bosschere, Koen (Hrsg.): *Compiler Construction* Bd. 7791. Springer Berlin Heidelberg, 2013, 102-122

[6] Braun, Matthias ; Buchwald, Sebastian ; Zwinkau, Andreas: Firm—A Graph-Based Intermediate Representation / Karlsruhe Institute of Technology. Version: 2011. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470. Karlsruhe, 2011 (35). – Forschungsbericht

[7] Chambers, Craig ; Dean, Jeffrey ; Grove, David: Frameworks for intra- and interprocedural dataflow analysis / Department of Computer Science and Engineering University of Washington. 1996. – Forschungsbericht

[8] Choi, Jong-Deok ; Cytron, Ron ; Ferrante, Jeanne: Automatic Con-

struction of Sparse Data Flow Evaluation Graphs. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1991 (POPL '91). – ISBN 0–89791–419–8, 55–66

[9] Click, Cliff ; Cooper, Keith D.: Combining Analyses, Combining Optimizations. In: *ACM Transactions on Programming Languages and Systems* 17 (1995), March, Nr. 2, S. 181–196

[10] Click, Cliff ; Paleczny, Michael: A simple graph-based intermediate representation. In: *SIGPLAN Not.* 30 (1995), März, Nr. 3, 35–49. `http://dx.doi.org/10.1145/202530.202534`. – DOI 10.1145/202530.202534. – ISSN 0362–1340

[11] Cooper, Keith D. ; Harvey, Timothy J. ; Kennedy, Ken: *A Simple, Fast Dominance Algorithm*. 2001

[12] Cooper, Keith D. ; Harvey, Timothy J. ; Kennedy, Ken: Iterative data-flow analysis, revisited / Department of Computer Science Rice University. 2002. – Forschungsbericht

[13] Cooper, Keith D. ; Harvey, Timothy J. ; Kennedy, Ken: An empirical study of iterative data-flow analysis. In: *CIC* (2006), S. 266 – 276

[14] Corporation, Standard Performance E.: *CPU2000 benchmark*. `http://www.spec.org/cpu2000/`. `http://www.spec.org/cpu2000/`. Version: 2000

[15] Dwyer, Matthew B. ; Clarke, Lori A.: A Flexible Architecture for Building Data Flow Analyzers. In: *Proceedings of the International Conference on Software Engineering* Bd. 17, 1998, S. 554–564

[16] Fietz, Jonas: *Deriving Restrictions on Value Types*, Karlsruher Institut für Technologie (KIT), Studienarbeit, Juli 2010

[17] Hall, Mary W. ; Mellor-Crummey, John M. ; Carle, Alan ; Rodríguez, René G.: FIAT: A Framework for Interprocedural Analysis and Transformation. In: *Proceedings of the sixth workshop on languages and compilers for parallel computing*, Springer-Verlag, 1995, S. 522–545

[18] Harrison, W.: Compiler Analysis of the Value Ranges for Variables. In: *IEEE Transactions on Software Engineering* 3 (1977), Mai, S. 243 – 250

[19] Hecht, Matthew S.: *Flow Analysis of Computer Programs*. New York, NY, USA : Elsevier Science Inc., 1977. – ISBN 0444002162

[20] Hind, Michael ; Pioli, Anthony: Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In: *Lecture Notes in Computer Science*, Springer-Verlag, 1998, S. 57–81

[21] Kam, John B. ; Ullman, Jeffrey D.: Global Data Flow Analysis and Iterative Algorithms. In: *Journal of the Association for Computing Machinery* 23 (1976), S. 158–171

[22] Kam, John B. ; Ullman, Jeffrey D.: Monotone Data Flow Analysis Frameworks. In: *Acta Informatica* 7 (1977), S. 305–317

[23] Kildall, G.A.: *Global Expression Optimization During Compilation*. University of Washington, 1972

[24] Lengauer, Thomas ; Tarjan, Robert E.: A fast algorithm for finding dominators in a flowgraph. In: *ACM Trans. Program. Lang. Syst.* 1 (1979), Januar, Nr. 1, 121–141. http://dx.doi.org/10.1145/357062.357071. – DOI 10.1145/357062.357071. – ISSN 0164–0925

[25] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM / University Karlsruhe. Version: September 2002. http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps. Universität Karlsruhe, Fakultät für Informatik, September 2002 (2002-5). – Forschungsbericht. – 75 S.

[26] Marlowe, Thomas J. ; Ryder, Barbara G. ; Burke, Michael G.: Defining Flow Sensitivity in Data Flow Problems / IBM T. J. Watson Research Center. 1995. – Forschungsbericht

[27] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, 1999. – ISBN 978–3–642–08474–4

[28] Patterson, Jason R. C.: Accurate static branch prediction by value range propagation. In: *SIGPLAN Not.* 30 (1995), Juni, Nr. 6, 67–78. http://dx.doi.org/10.1145/223428.207117. – DOI 10.1145/223428.207117. – ISSN 0362–1340

[29] Quinlan, Daniel J.: ROSE: Compiler Support for Object-Oriented Fra-

meworks. In: *Proceedings of Conference on Parallel Compilers (CPC2000)*, 2000. – http://rosecompiler.org/

[30] Seltenreich, Andreas: *Bitbreitenminimierung per Datenflussanalyse*, Karlsruher Institut für Technologie (KIT), Studienarbeit, Februar 2013. http://pp.info.uni-karlsruhe.de/uploads/publikationen/seltenreich13studienarbeit.pdf

[31] Tjiang, Steven W. K. ; Hennessy, John L.: Sharlit - a tool for building optimizers. In: *SIGPLAN Not.* 27 (1992), Juli, Nr. 7, 82–93. http://dx.doi.org/10.1145/143103.143120. – DOI 10.1145/143103.143120. – ISSN 0362–1340

[32] Tok, TeckBok ; Guyer, SamuelZ. ; Lin, Calvin: Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. Version: 2006. http://dx.doi.org/10.1007/11688839_3. In: Mycroft, Alan (Hrsg.) ; Zeller, Andreas (Hrsg.): *Compiler Construction* Bd. 3923. Springer Berlin Heidelberg, 2006. – DOI 10.1007/11688839_3. – ISBN 978–3–540–33050–9, 17-31

[33] Wegman, Mark N. ; Zadeck, F. K.: Constant Propagation with Conditional Branches. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Nr. 2, S. 181–210