

Kombinierte Datenfluss-Analysen

Fridolin Häuser

12. Juni 2008

Hauptseminar Programmanalysen

Zusammenfassung

Gegenseitiger Informationsaustausch kann in der Optimierung von Quellcode ein besseres, d.h. leistungsstärkeres Ziel erreichen. Die Kombination von Datenfluss-Analysen mit verschiedenen Analysen beziehungsweise Optimierungen kann durch unterschiedliche Modelle realisiert werden. Meine Arbeit verfolgt grundsätzlich drei unterschiedliche Ansätze: Zunächst beschreibt die sequentielle Lösung das Hintereinanderschalten der einzelnen Analysen. Die Analysen werden nacheinander durchlaufen und gegebenenfalls wird im gesamten Prozess iteriert. Einzelne individuell angepasste Algorithmen verschmelzen die elementaren Analysen. Die letzte vorgestellte Möglichkeit ist ein System, das den Zusammenbau einzelner, modular gestalteter und einfach anzupassender Datenfluss-Analysen ermöglicht.

1. Einführung

Um die notwendigen Begrifflichkeiten, Notationen und Voraussetzungen zu schaffen, gehe ich zuerst auf die allgemeine Thematik der Datenfluss-Analysen ein und präsentiere einige grundlegende Punkte. Die Arbeit zeigt die benötigten Graphen, Verbünde und einige im Weiteren verwendete elementare Datenflussanalysen auf.

Im Hauptteil dieser Ausarbeitung stelle ich drei Arten dar, mit denen die Kombination der Datenfluss-Analysen durchgeführt werden kann. Die sequentielle Ausführung der verschiedenen Analysen ist die einfachste Option, die auch einen kleinen Ausblick in die Phase Ordering-Problematisierung bietet.

Die zweite Kombinationsmöglichkeit arbeitet mit Hilfe von handgeschriebenen Algorithmen, dabei dient der Algorithmus von Wegman und Zadeck [12] als Beispiel.

Die dritte Vorgehensweise basiert auf der Grundlage des Systems von Lerner, Grove und Chambers [10]. Sie gestalten die Verbindung unterschiedlicher Analysen modular und benutzen dabei einen weiteren Graphen, der die Kommunikation unter den elementaren Einzelteilen ermöglicht. Ein anderer Ansatz wäre die Kombination verschiedener

Analysen über größere Tabellen.

Nach der Präsentation der einzelnen Lösungswege stelle ich noch kurz die interprozedurale Optimierung vor. In einem weiteren Ausblick stelle ich die Einflussmöglichkeiten dieses Bereichs auf die Kombination der Analysen dar.

1.1 Datenfluss-Analysen

Datenfluss-Analysen können auf Programmcode angewandt werden, um Verkürzungen oder Laufzeitverbesserungen des Programms zu erzielen. Zum Beispiel können einige Zuweisungen bei der Analysetechnik Konstanten Propagierung eingespart werden.

Die verwendeten Algorithmen basieren alle auf statischen Analysen. Dynamische Analysetechniken werden nicht weiter betrachtet.

Dieses Dokument verwendet in seinen Beispielen hauptsächlich folgende Formen der Datenfluss-Analyse:

Klassenanalyse: Diese Analyse soll die Klasse des Objekts einer Variable zurückgeben, um daraus weitere Schlüsse zu ziehen. Dieses Vorgehen kann - in einem späteren Beispiel - einen Funktionsaufruf in einer Vererbungshierarchie entschlüsseln.

Inlining: Der Funktionsinhalt selbst ersetzt bei dieser Technik die Funktionsaufrufe. Dabei ist es die Bestrebung dieser Technik, Laufzeitgeschwindigkeit auf Kosten der Programmgröße zu gewinnen. Ziel des Inlining sind oft auch Optimierungen anderer Analysen. Durch die Ersetzung der Funktion kann möglicherweise eine weitere Optimierung auf den Programmcode angewandt werden, was dann die Effizienz des gesamten Codes stark verbessert. Später verwende ich ein Beispiel, in dem das Inlining den Funktionsaufruf mit einer einzelnen Zeile ersetzen kann und somit die Geschwindigkeit und die Übersichtlichkeit deutlich erhöht.

Konstanten Propagierung: In diesem Dokument fassen wir Konstantenfaltung (Constant Folding) und Konstanten Propagierung (Constant Propagation) unter

diesem Schlagwort zusammen. Dies erleichtert den Umgang mit der Begrifflichkeit. Unter Konstantenfaltung versteht man das Zusammenfassen eines konstanten Ausdrucks zur Compilezeit. Dadurch kann die Berechnung des Ausdrucks zur Laufzeit eingespart werden. Konstanten Propagierung beschäftigt sich dagegen mit der Propagierung konstanter Werte einer Variable. Auch diese Optimierung findet natürlich zur Compilezeit statt. Diese Technik kann auch andere Vereinfachungen und Optimierungen denkbar machen oder vereinfachen. In folgenden Beispielen zeigen wir, wie das Zusammenspiel dieser Analysetechnik und der Elimination von totem Code verschiedene Programmfragmente vereinfacht und optimiert.

Conditional Branches: Dieser Begriff steht für eine Art der Elimination von totem Code, die ihren Einsatz bei einem bedingten Sprung findet. D.h. bei einem if-Statement wird entschieden, ob alle Zweige erreichbar sind. Wenn die Bedingung des Statements immer zutrifft, wäre der else-Zweig der Anweisung genauso wie die eigentliche Bedingungsabfrage überflüssig und man kann sie durch einen kürzeren und besseren Ausdruck ersetzen.

Elimination von totem Code: Hierbei handelt es sich um eine Kontrollflussanalyse, jedoch ist sie mit verschiedenen Datenfluss-Analysen kombinierbar. Ziel dieser Analysetechnik ist es Programmcode zu finden, der niemals ausgeführt wird, und diesen zu entfernen. Dabei arbeitet diese Technik, um ein sinnvolles Ergebnis zu erzielen, im mindesten Fall auf dem Kontrollflussgraphen. Die Suche nach Variablen, die niemals Verwendung finden, ist ein weiterer wichtiger Punkt. Manchmal ist es hilfreich diese Variablen aus Gründen der Übersichtlichkeit komplett aus dem Code zu entfernen. Des Weiteren kann sich eine Rückmeldung an den Programmierer ebenfalls vorteilhaft auswirken. (Unreachable Code Elimination)

Der folgende Algorithmus nutzt den Static Single Assignment-Graph, er spielt für den Ansatz von Wegman und Zadeck [12] eine grundlegende Rolle. (kurz: SSA-Graph) Der Graph zeichnet sich dadurch aus, dass jede Variable genau einmal zugewiesen wird und eine Auftrennung der existierenden Variablen über Indizes stattfindet.

Eine Realisierung des Algorithmus von Wegman und Zadeck [12] ist mit Hilfe von Def-Use Chains zu erreichen, allerdings kann diese Änderung das Ergebnis negativ beeinflussen. Einige Treffer, die mit der SSA-Form gefunden werden, sind nicht durch eine Implementierung über Def-Use Chains aufzeigbar.

Monotonität der Analysefunktionen ist ein sehr wichtiger Faktor in der Betrachtung. Besitzen verschiedene Analysefunktionen diese Eigenschaft nicht, kann kein fester

Punkt (fixed point) in der kompletten Analyse garantiert werden. Diese Tatsache führt zu Problemen mit der Terminierung und der Ergebnisfindung der einzelnen Algorithmen. Erst wenn dieser Fixpunkt innerhalb einer einzelnen Untersuchung erreicht wurde, kann die gesamte Analyse fortfahren. Zwar erläutere ich die genaue Auswirkung der Monotonität an den entsprechenden Stellen nochmals näher, muss aber direkt darauf hinweisen, dass die Monotonität bei der Kombination verschiedener Analysen nicht durch die Monotonität der Elemente gewährleistet werden kann. Es ist erforderlich eine monotone Eigenschaft der Gesamtfunktion in diesem Fall neu zu zeigen und zu beweisen.

Man nutzt für die Speicherung der gewonnenen Informationen die Struktur des Verbunds (Lattice). Die gespeicherte Information und die monotone Eigenschaft stellt fest, wann der Fixpunkt erreicht ist. Verändern sich in einer gesamten Iteration der betrachteten elementaren Analyse keine gespeicherten Werte, so ist die Analyse durch ihre monotonen Eigenschaften an ihrem Fixpunkt angekommen. Die Gesamtfunktion kann fortfahren, sobald dieser Teil abgeschlossen ist.

Am Ende der Beschreibung des Systems von Lerner, Grove und Chambers [10] findet eine Diskussion ihrer experimentellen Laufzeitergebnisse statt. Diese verschaffen einen Überblick über das allgemeine Laufzeitverhalten.

1.2 Meet Operator

Für das weitere Verständnis ist die Notation der erhaltenen Informationen wichtig. Informationen werden im Lattice (Verbund) festgehalten. Diese Struktur liegt den einzelnen Analysen wie auch den gesamten Analysen zu Grunde. In ihr werden die gelesenen Informationen festgehalten. Zum Beispiel wird bei der Konstanten Propagierung die Zuweisung $x := 10$; im Verbund eine Zuordnung von $X \rightarrow 10$ hinterlassen. Bei Gewinnung weiterer Informationen muss der Umgang festgelegt werden. Leider unterscheiden sich die Schreibweisen in den Dokumenten von Wegman und Zadeck [12] im Gegensatz zum Dokument von Lerner, Grove und Chambers [10]. Das Symbol \top wird für die Darstellung aller möglichen Verhalten des Programms und \perp für kein Verhalten des Programms verwendet. Damit benutzen wir die Notation von Lerner, Grove und Chambers [12]. Der Algorithmus startet in optimistischer Weise mit der Initialisierung aller Anfangswerte mit \perp . Dabei gilt es zu beachten, dass ein vorzeitiger Abbruch in diesem Fall zu einer fehlerhaften Anpassung führen kann. Eine Initialisierung mit \top kann zwar zur Fehlervermeidung bei einem unplanmäßigen Halt führen, jedoch kann die Effizienz der Algorithmen eingeschränkt werden.

Um den Umgang mit einer gespeicherten Information und hinzukommenden neuen Daten festzulegen, benutzen wir folgende Tabelle:

$\text{any} \sqcap \perp$	$= \text{any}$
$\text{any} \sqcap \top$	$= \top$
$c \sqcap c$	$= c$
$c_1 \sqcap c_2$	$= \top$ if $c_1 \neq c_2$

Tabelle 1. Meet Operator

2. Sequentielle Kombination

Diese Technik führt verschiedene Analysen hintereinander aus. Über ihre einzelnen Elemente kann dabei auch iteriert werden, um bessere Ergebnisse zu erzielen. Diese triviale Vorgehensweise für die Kombination verschiedener Datenfluss-Analysen ist in ihrer Überlegung einfach. Die endgültige Kombination besitzt dabei nur einen kleinen Implementierungsaufwand.

2.1 Ablauf

Für die sequentielle Kombination müssen die einzelnen Datenfluss-Analysen über einen monotonen Funktionsverlauf verfügen. Ansonsten kann der Fixpunkt-Ansatz nicht sicher benutzt werden.

Die einzelnen Analysen werden sequentiell ausgeführt und die Ergebnisse der Einzelanalysen werden direkt im Programmcode umgesetzt und als Zwischenergebnis gespeichert. Damit wenden wir die nachfolgenden Phasen direkt auf den optimierten Code an. Die einzelnen Funktionen können erst voneinander profitieren, wenn ein Endergebnis in der elementaren Untersuchung vorliegt. Analyseinternen Zwischenergebnissen und wechselseitigen Zusammenspielen der Durchgänge können leider sequentiell nicht von einander profitieren. Der nächste Teil der gesamten Funktion kann erst starten, wenn eine Phase vollständig abgeschlossen ist. Man kann einzelne Analysen mehrfach durchlaufen, um so eine längere Iteration innerhalb der Kombination zu starten.

Eine entscheidende Rolle für die Qualität des Ergebnisses spielt dabei die Reihenfolge der Ausführung. Nicht nur in Fragen der Geschwindigkeit, sondern auch in der Länge des Ergebniscodes können starke Unterschiede auftreten. Dieses Problem bezeichnet man als Phase Ordering-Problem, ich erläutere es im nächsten Abschnitt.

2.2 Phase Ordering-Problem

Wie bereits erwähnt, ist die Reihenfolge der einzelnen Optimierungsphasen entscheidend für die gesamte Effektivität der Optimierung.

Länge und Geschwindigkeit des Ergebniscodes sind Kriterien. Die unterschiedlichen Permutationen der Optimierungsreihenfolge können diese Eigenschaften des Ergebniscodes stark beeinflussen. Die bestmögliche Permutation ist dabei nicht allgemeingültig festlegbar. Die folgenden drei Aspekten spielen bei der Bestimmung eine Rolle:

- Die Zielplattform
- Die exakt verwendeten Analysetechnik, d.h. im weitesten Sinne der Compiler selbst
- Der Quellcode des Programms

Die Menge von verschiedenen Reihenfolgen im Phase Ordering-Problem kann gewaltig sein. Selbst ohne die Nutzung iterativer Analysesequenzen, befindet man sich bei 15 Optimierungsphasen bei 15! Kombinationen. Deutlich bessere Ergebnisse erzielt man, indem man zusätzliche Iterationen in die Phasen einbaut. Kulkarni [8] erreichte bei seinen Experimenten mit dem VPO-Compiler mit insgesamt 15 einzelnen Optimierungsphasen eine optimale Sequenz von 15⁴⁴ Anordnungsmöglichkeiten der einzelnen Phasen im experimentellen Gebrauch.

Diese Größenordnung lässt vermuten, dass die Suche nach der optimalen Phase mit der Nutzung von brute-force Algorithmen ineffizient sein wird. Kulkarni zeigte einen Lösungsweg, mit dem man eine solche Suche realisieren kann. Dieses Vorgehen funktioniert durch Ausschluss verschiedener Anordnungen über Plausibilisierung und Vergleiche ihrer Reihenfolgen. Bei ähnlichen Voraussetzungen wird die zu untersuchende Menge gekürzt.

Da dieser Ansatz auch für größere Funktionen zu langen Laufzeiten führt, und dabei nicht den intuitiven Ansatz zur Problemlösung darstellt, verwendet man in der Praxis heuristische Suchverfahren. Zum Beispiel können Verfahren wie „Hill Climbing“ oder der genetische Algorithmus aus dem Bereich der Evolutionären Algorithmen verwendet werden.

Nach Kulkarni, Whalley und Tyson [7] müssen die Verfahren selbst keine besondere Komplexität besitzen um ordentliche Ergebnisse zu erzeugen. Allerdings kann durch die Konzentration der Algorithmen auf die Blätter der Ordnungsfunktionen ein günstiges und akzeptables Ergebnis erzielt werden. Man kann die Kosten dieser Suche deutlich reduzieren, da die zu untersuchenden Blätter vom Gesamtumfang etwas weniger als 5% ausmachen.

Dieses Problem tritt nicht nur bei der sequentiellen Kombination auf, sondern auch bei der Verwendung anderer Möglichkeiten. Wenn eine in das System unmöglich einzufügende Optimierung erwünscht ist, muss diese vorhergehend oder nachträglich ausgeführt werden. Dabei tritt natürlich wieder das Phase Ordering-Problem auf. Das bedeutet, dass sich nicht immer alle Optimierungsmöglichkeiten

in die Algorithmen oder Frameworks einarbeiten lassen. Allerdings können diese in einer weiteren Phase dazugeschaltet werden, was natürlich eine sequentielle Kombination bedeutet und damit das vorgestellte Problem hervorruft.

2.3 Beispiele

Die zwei folgenden Beispiele zeigen zum einen einen Fall, in dem die sequentielle Kombination funktioniert und zum anderen einen Fall, in dem sie wirkungslos auf das Programmfragment reagiert. Das zweite Beispiel wird im Folgenden noch für andere Techniken aufgegriffen, um deren verbesserte Möglichkeiten zu demonstrieren.

```
x := 10;
y := x;
if (y == 10) {
    DoSomething();
} else {
    DoSomethingElse();
}
```

Wir wenden die Analysen Konstanten Propagierung und Elimination von totem Code auf diesen Code in aufeinanderfolgender Reihenfolge an und behalten dabei die Aufzählungsfolge als Ausführungsreihenfolge bei. Würden wir die Sequenz in anderer Reihenfolge ausführen, müssten wir einen weiteren Durchlauf anfügen, da die zweite Untersuchung das Ergebnis der ersten benötigt. Wie bereits ausgeführt, ist die Bestimmung dieser optimalen Reihenfolge schwierig.

Zuerst ordnet der Algorithmus X den konstanten Wert zu und propagiert diesen weiter. Nachdem diese Tatsache bekannt ist, wird Y ebenfalls auf den konstanten Wert 10 gesetzt. Die Bedingung der if-Abfrage kann durch die Anpassung nun ebenfalls mit „wahr“ festgelegt werden. Nach diesen zwei Ersetzungen hat die erste Analyse einen fixen Punkt erreicht und es kommt zu keiner weiteren Veränderung in ihren Iterationen.

Mit der Eliminierung des unerreichbaren Codes wird die Gesamtanalyse fortgesetzt. Diese wird feststellen, dass der else-Zweig des if-Statements niemals erreicht wird und damit toter Code ist. Der Grund dafür ist natürlich die grundsätzlich wahre Bedingung der if-Anweisung. Diese Teile des Programmfragments können entfernt werden, und auch diese Analyse erreicht ihren Fixpunkt und kann damit abschliessen.

Weitere Iterationen der elementaren Analysen bergen keine Veränderung des Programmfragments und sind in diesem Beispiel unnötig. Wäre - wie bereits erwähnt - die Analysereihenfolge anders gewählt, dann müsste eine Iteration in der Gesamtanalyse eingebaut werden, um das oben genannte Ergebnis zu erreichen.

```
X := 10;
while (...) {
    if (x == 10) {
        DoSomething();
    } else {
        DoSomethingElse();
        X := X + 1;
    }
}
Y := X;
```

Dieses Beispiel entstammt einem Bericht von Lerner, Grove und Chambers [10]. Es wird auch im Weiteren zur Beschreibung der anderen Vorgehensweisen verwendet. Damit will ich aufzeigen, wie direkt kombinierte Analysen einen besseren und effizienteren Zugang für die Programoptimierung bieten. Die sequentielle Kombination ist zwar intuitiver und einfacher Einstieg, sie unterstützt jedoch die analytischen Zusammenspiele nicht direkt. Dies verhindert sie durch ihre Abgrenzung zwischen den Untersuchungen.

Eine interessante Frage für dieses Beispiel ist natürlich, warum die sequentielle Ausführung der einzelnen Analysen keine Veränderung des Codefragments hervorruft. Es kommt in gewisser Weise auch auf die Implementierung des eingesetzten Konstanten Propagierung-Algorithmus an. Nimmt man jedoch an, dass keine zusätzlichen Eigenschaften eingefügt wurden, sondern der Algorithmus simpel umgesetzt wurde, kann keine Veränderung hervorrufen werden. Eine naive Implementierung der Elimination von totem Code ist in diesem Fall machtlos, da die Konstanz in der Sprungbedingung nicht festgestellt ist.

Zur näheren Erläuterung gehe ich auf den Verlauf der Analyse ein. Die elementare Analyse kann nicht feststellen, ob und in wie weit sich der Wert von X innerhalb der Schleife modifiziert. Deswegen wird der Meet-Operator durch die Verschmelzung des ersten und zweiten Durchlaufs zwischen 10 und 9 ein \top produzieren. Dieses \top kann durch eine weitere Meet-Operation nicht weiter beeinflusst werden und bleibt in allen Durchläufen erhalten. Dadurch kann man keine Aussage über den Wahrheitsgehalt der Bedingung treffen.

Beide einzelnen Analysen haben ohne einen gegenseitigen Datenaustausch während des Durchlaufs keine Chance, das vorliegende Programmstück zu verkürzen oder anzupassen. Die Schleife in Kombination mit dem if-Statement erschwert die Anpassung in diesem Fall.

2.4 Laufzeit

Die benötigte Compilezeit bei der Anwendung einer sequentiell angeordneten Kombination ist vergleichsweise gesehen sehr groß. Bis jede einzelne Funktion in ihrer letzten Iteration den Fixpunkt erreicht hat, kann über die gesamte

Menge der Analysen iteriert werden, um von den gegenseitigen Endergebnissen zu profitieren. Gibt es also in einer einzelnen Phase eine Änderung, muss man nochmals alle Analysen durchlaufen, um zu garantieren, dass auch wirklich das Endergebnis erreicht wurde. Wird nicht iteriert, muss man zwar jede Analyse nur ein einziges Mal durchführen, jedoch treten in deutlich massiertere Form Phase Ordering-Probleme auf. Zusätzlich wird das Endergebnis von deutlich schlechterer Qualität sein.

3. Kombination zu einem gesamten Algorithmus

Mit diesem Ausdruck sind Kombinationstechniken gemeint, die verschiedene atomare Analysen in einer Ausführung vollziehen. Die einzelnen Analysen können in keiner Form alleinstehend durchgeführt werden, ohne die Funktionalität des geschaffenen Algorithmus zu verhindern.

Die so geschaffenen Algorithmen sind von Hand konstruiert worden, um einen besseren Profit aus dem gegenseitigen Zusammenspiel zu gewinnen. Diese Technik kann dem Phase Ordering-Problem beikommen.

Verschiedene Algorithmen sind zu diesem Thema verfügbar. Zum Beispiel geben Chambers und Ungar [2] einen Algorithmus für das Inlining, die Klassenanalyse und das Splitting an. Pioli und Hind [11] dagegen haben einen Algorithmus erschaffen, der die Methoden der Konstanten Propagierung und Pointer Analysen zusammenfaßt.

Um die Überlegungen zu veranschaulichen, verwende ich den Algorithmus von Wegman und Zadeck [12]. Die Bedeutung ihrer Ausarbeitung wird auch über die Menge der Zitierungen in anderen Publikationen angedeutet, insgesamt 92 Zitate stellt die ACM-Seite fest. Ihr Algorithmus heißt Sparse Conditional Constant (kurz: SCC) und basiert auf der SSA-Form. Er wurde mit Hilfe verschiedener anderer Algorithmen erstellt, an deren Anfang eine Ausarbeitung von Kildall [6] steht. Diese Algorithmen sind meist direkter und ihnen fehlt eine geschickte Implementierung, um Geschwindigkeit hinzuzugewinnen. Schon seine Entstehungsgeschichte zeigt die Komplexität der Erstellung eines solchen Algorithmus. Dabei werden nur zwei elementare Analysetechniken auf der Struktur der SSA-Form zusammengefasst, nämlich Konstanten Propagierung und Conditional Branches.

3.1 Sparse Conditional Constant

Der Sparse Conditional Constant-Algorithmus vereinigt die Konstanten Propagierung und die Elimination von totem Code. Er basiert dabei auf zwei Arbeitslisten:

FlowWorkList: Besteht aus den Kanten des Programmflussgraphen und ist mit den Kanten des Startknotens initialisiert

SSAWorkList: Besteht aus Kanten des SSA-Graphen und ist am Anfang leer

Zusätzlich zu diesen zwei Listen bekommt jede Kante des Programmfluss-Graphen einen *Wahrheitswert* zugewiesen. Dieser ist am Anfang überall auf „falsch“ gesetzt und entspricht der Ausführbarkeit des Ausdrucks während des Programmverlaufs. Der Verbund sollte dem ganzen zu Grunde gelegt sein, um die Informationen zu nutzen und zu verstehen. Die Inhalte werden zu Beginn des Algorithmus in optimistischer Form auf \perp gesetzt.

Mit der Festlegung der benötigten Struktur kann ich auf den groben Ablauf mit einem kleinen Beispiel eingehen. Erst nach dieser kurzen Präsentation findet die Vorstellung des exakten Algorithmus statt. Im Anschluss an diese Darstellung gehe ich nochmals auf ein Beispiel zur Verdeutlichung ein.

```
x := 10;
y := x;
if (y == 10) {
    DoSomething();
} else {
    DoSomethingElse();
}
```

Die Wahrheitswerte der Kanten im Programmflussgraphen sind auf „falsch“ gesetzt. Nur die Kante zur Zuweisung $x:=10$; ist in der entsprechenden *FlowWorkList* enthalten. Die erste Kante wird aus der Liste entfernt und der zugehörige Wahrheitswert wird auf „wahr“ gesetzt. Der Algorithmus propagiert die im Ausdruck enthaltene Information über die entsprechenden SSA-Kanten. Der Verbund hält somit die Zuordnung $x=10$ an der Stelle $y:=x$; fest. Der Wahrheitswert beinhaltet die Tatsache, dass man die Kante betrachtet hat und damit die Möglichkeit, dass der Programmverlauf sie beschreitet. Knoten, die nach Abschluss des Algorithmus keine Eingangskante mit „wahr“ besitzen, sind also toter Code.

Da es nur eine Kante für den weiteren Verlauf gibt, wird die nächste Kante im Programmflussgraphen in die *FlowWorkList* eingefügt. Direkt danach betrachtet man diese Kante wieder und reicht über die SSA-Kante den Wert $y=10$ an die Bedingung weiter. Zusätzlich setzt der Algorithmus den zugehörigen Wahrheitswert wieder auf „wahr“.

Danach arbeitet der Algorithmus an dem if-Statement weiter. Die Information $y=10$ liegt durch die betrachteten SSA-Kanten vor. Nur der „wahre“-Zweig des Programmfragments wird in die *FlowWorkList* eingefügt. Wäre der Inhalt des Verbunds \top , dann müsste man beide Wege hinzufügen. Ich gebe hier einen kurzen Ausblick auf das spätere Beispiel. Dabei sollte beachtet werden, dass bei einer Schleifenverschachtelung gegebenenfalls auch der alternative Weg zu einem späteren Zeitpunkt in die *FlowWork-*

List eingefügt wird. Dies geschieht mit Hilfe der Propagierung der Informationen über die SSA-Kanten. Eine neue Betrachtung dieser Form entsteht nur unter der Bedingung, dass eine Eingangskante des Bedingungs-Knotens im Programmflussgraphen auf „wahr“ gesetzt ist.

Der weitere Verlauf des Beispiels setzt den Inhalt des gewählten Zweigs des if-Statements auf „wahr“ und beendet anschließend den Algorithmus, da keine weiteren Kanten in den Listen enthalten sind. Im Ergebnis sind damit die nicht verwendeten Kanten des Programmfragments auf „falsch“ gesetzt. Des Weiteren speichert der Verbund die Informationen, die die Konstanten Propagierung benötigt.

3.2 Ablauf

Das genaue Vorgehen sieht wie folgt aus: Enthalten die beiden Listen keine weiteren Elemente mehr, dann wird die Ausführung terminiert. Im anderen Fall wird eine Kante aus der Liste genommen.

Der Algorithmus betrachtet die gewählte Kante. Handelt es sich dabei um eine Kante des Programmflussgraphen, dann wird, wenn der Wahrheitswert „wahr“ ist, keine weitere Betrachtung stattfinden. Ansonsten führt man folgende Schritte aus:

- Setze den Wahrheitswert auf „wahr“
- Führe die Funktion $Visit-\phi$ für alle ϕ -Funktionen im Zielknoten der Kante des Programmflussgraphen aus
- Beim ersten Besuch des Knoten soll die Funktion $VisitExpression$ ausgeführt werden (Wenn nur eine betretende Kante auf Wahr gesetzt ist, wurde der Knoten das erste Mal besucht)
- Wenn nur eine einzelne ausgehende Kante existiert, wird diese der $FlowWorkList$ hinzugefügt

Ist die zu betrachtende Kante nun aus der $SSAWorkList$ und ist in ihrem Zielknoten eine ϕ -Funktion, dann soll $Visit-\phi$ ausgeführt werden. Ist das Ziel hingegen ein Ausdruck, dann untersucht der Algorithmus alle Wahrheitswerte von den Kanten, die den Knoten des Ausdrucks erreichen. Ist einer dieser Wahrheitswerte „wahr“, dann soll $VisitExpression$ ausgeführt werden. Ansonsten findet keine weitere Aktion statt. Der Algorithmus nach Wegman und Zadeck gibt keine Vorschrift an, mit Hilfe der man entscheiden kann, welche Kanten zuerst abzuarbeiten sind.

Ich stelle nun die zwei angesprochenen und benötigten Funktionen vor. Die betrachteten Variablen sind die zugehörigen Variablen im SSA-Graphen. Dabei funktioniert $Visit-\phi$ wie folgt:

Im Programmflussgraphen betrachtet man die zu der betreffenden Variablendefinition gehörenden SSA-Kante. Ist diese ausführbar, d.h. ist der Wert ihrer Markierung auf

„wahr“ gesetzt, wird der Variable der zugehörige Wert zugeordnet. In einem anderen Fall wird der Wert \perp für diese Variable festgesetzt.

Die Funktion $VisitExpression$ funktioniert wie folgt:

Die Tabelle zum Meet-Operator beschreibt die Endinformation, die berechnet wird, wenn verschiedener Werte bei den einzelnen Operanden vorliegen. Damit kann sich der Wert der Variable verändern. Wenn dies geschieht, dann müssen folgende Punkte ausgeführt werden:

1. Wenn der Ausdruck ein Teil eines Zuweisungsknotens war, dann müssen alle SSA-Kanten zur $SSAWorkList$ hinzugefügt werden, die von dieser Definition aus starten.
2. Wenn der Ausdruck einen Sprung kontrolliert, müssen entsprechend der vorhandenen Informationen über den Ausdruck Kanten zur $FlowWorkList$ hinzugefügt werden. Ist der Wert am Ausdruck \top , dann kann der Algorithmus keine Aussage über den weiteren Verlauf treffen und er fügt alle ausgehenden Kanten zur $FlowWorkList$ hinzu.

3.3 Beispiel

Um den Algorithmus näher zu beschreiben, greifen wir nun das vorangegangene Beispiel auf. Es zeigt sich, dass das gegebene Programmfragment deutlich zusammenschrumpft und beide erwünschten Analysen auf ihm durchgeführt wurden. Damit findet dieser Algorithmus Vorkommen, die bei sequentieller Vorgehensweise nicht ersetzt werden konnten.

```
X := 10;
while (...) {
  if (x == 10) {
    DoSomething();
  } else {
    DoSomethingElse();
    X := X + 1;
  }
}
Y := X;
```

Man fügt die Kanten der Schleife mit Hilfe der ersten Zuweisung in die Liste ein. Zusätzlich hält der Algorithmus fest, dass der Wert von X konstant zehn beträgt. Dies geschieht über die Ausführung der Funktion $Visit-\phi$. Am Schleifenbeginn werden durch die Funktion $VisitExpression$ sowohl die Kante zum If-Statement wie auch die Kante zu $Y:=X$; hinzugefügt, da keine eindeutige Entscheidung getroffen werden konnte. Dieses Vorgehen geschieht unter der Annahme, dass die Bedingung der Schleife \top gesetzt ist und damit unbestimmbar bleibt. Damit erreichen wir den Sprung am if-Statement oder können direkt mit der nächsten Kante fortfahren.

An der Sprungstelle des If-Statement ist X noch immer der Wert „zehn“ zugeordnet, da die Zuweisung $X := X + 1$;

noch nicht ausführbar ist. Da es sich um den ersten Besuch dieser Kante handelt, wird *VisitExpression* ausgeführt. Dabei wird ein konstanter Wahrheitswert erkannt und nur die Kante zum Knoten vom Ausdruck `DoSomething()`; wird in die *FlowWorkList* eingefügt.

Im Beispiel wird davon ausgegangen, dass dieser Funktionsaufruf den weiteren Verlauf des Programms nicht verändert. Sie wird nur markiert und es geht bei der ursprünglichen Liste weiter. In dieser befindet sich nur noch die Kante `X := Y`;

Die Verfolgung der SSA-Kanten zeigt auf, dass sie den Wert „zehn“ beinhalten muss, da alle anderen Zuweisungen von X nicht weiter ausführbar sind. Selbst wenn die als Letztes betrachtete Zuweisung nun vor dem Schleifeninhalt angesehen wird, findet keine andere Informationsgewinnung statt. Die Änderung der Reihenfolge spielt keine Rolle, da die Inhalte der Schleife noch komplett auf „falsch“ gesetzt sind und somit nachgereicht würden, sobald die Funktion *Visit-Expression* ausgeführt wird.

Der Algorithmus kann damit das Programmfragment wie folgt zusammenfassen:

```
X := 10;
while (...) {
    DoSomething();
}
Y := 10;
```

Der unerreichbare Quellcode wurde aus dem Programm entfernt und der konstante Wert von X wurde weiterpropagiert. Die Optimierung beider elementarer Datenfluss-Analysen wurde also mit Hilfe dieses kombinierten Algorithmus durchgeführt.

3.4 Vergleich des Beispiels mit dem sequentiellen Vorgehen

Die Änderungen stehen im Gegensatz zum sequentiellen Vorgehen. Die sequentielle Analyse konnte in diesem Beispiel keine Änderung erzielen. Der Grund dafür war ihr fehlendes Zusammenspiel der einzelnen Phasen. Die Informationen von Zwischenergebnissen der Analysen konnten nicht weitergegeben werden.

Die Konstanten Propagierung hat keine Informationen über das Sprungverhalten des `if`-Statements und kann somit nicht feststellen, dass X mit einem festen Wert ersetzt werden kann. In der anderen Richtung kann keine Aussage ohne die Festsetzung von X auf einen konstanten Wert über das exakte Sprungverhalten getroffen werden. Beide Funktionen können dieses Programmstück in einer einfachen Implementierung nicht vereinfachen. Erst im Zusammenspiel mit einem gemeinsamen Algorithmus lässt sich eine passende Änderung erreichen. Dieses Vorgehen ermöglicht es,

den unbenutzten `if`-Zweig zu identifizieren und damit direkte Schlüsse im Rahmen der Analyse zu ziehen.

3.5 Theorie

Ein kurzer Ausblick auf einen mächtigeren, aber auch laufzeitintensiveren Algorithmus soll mit dem Hinweis auf Holley, Rosen [5] gegeben werden. Dieser besitzt im Gegensatz zum SCC eine exponentielle Laufzeit im Bezug auf die Graphen. Allerdings verzichte ich hier auf eine nähere Erläuterung.

SCC dagegen besitzt eine Laufzeit, die der Nummer von Kanten im Programmflussgraphen und im SSA-Graphen entspricht. Man besucht dabei die Kanten im Programmflussgraphen nur einmal und die des SSA-Graphen höchstens zweimal.

Damit verbindet dieses Vorgehen die zwei Datenfluss-Analysen Elimination von totem Code und Konstanten Propagierung in einem Algorithmus und bietet zusätzlich eine gute Laufzeit an. Das ordentliche Laufzeitverhalten solcher Analysen wird später nochmals in den experimentellen Ergebnissen von Lerner, Grove und Chambers festgestellt.

3.6 Bedeutung der gesamten algorithmischen Kombination

Das Problem der Erweiterbarkeit wird ersichtlich. Eine weitere direkte Kombination mit anderen Analysen und Optimierungsphasen ist schwierig, da die Integration direkt innerhalb der Funktion stattfinden muss und keine Modularität gegeben ist. Das bedeutet, dass eine weitere Kombination nur durch zusätzliche nicht automatisierte, eigene Arbeit möglich ist.

Bei der Verwendung einer solchen Kombinationstechnik, können im selben Durchlauf keine weiteren Analysen hinzugefügt werden. Ansonsten müsste der gesamte Algorithmus entsprechend angepasst und erweitert werden.

Allerdings können die beiden elementaren Analysen voneinander profitieren und den Programmcode in einem einzelnen, gesamten und überschaubareren Durchlauf optimieren. Bei einer sequentiellen Implementierung dieser Analyse-Kombination hätte man das gegebene Beispiel nicht auflösen können.

Diese Tatsache mit dem guten Laufzeitverhalten spricht für die Kombination verschiedener Analysen in Form einer gesamten Einzelfunktion.

Im Vergleich zu diesem Vorgehen versucht der zuletzt beschriebene Ansatz von Lerner, Grove und Chambers [10], der Effektivität dieses Ansatzes die Modularität hinzuzufügen. Dabei stellen sie die Einzelinformationen der Analysen leicht lesbar als Zwischengraph dar.

Erwähnenswert wäre dabei noch ein weiterer Ansatz von Click und Cooper [3]. Mit Hilfe dieses Modells können zu-

mindest drei Analysetechniken kombiniert werden (Global Value Numbering, Konstanten Propagierung und Eliminieren von totem Code). Leider wurden die Erweiterungsmöglichkeiten nicht weiter betrachtet. Deswegen können hier auch keine Rückschlüsse auf die allgemeine Modularität gegeben werden und wir beschäftigen uns mit der zuerst genannten Technik.

4. Ansatz von Lerner, Grove und Chambers

[10] Dieses Vorgehen zur Kombination von verschiedenen elementaren Analysen verspricht eine modulare und simple Erstellung der zusammengesetzten Funktion. Es ist ein allgemeiner Ansatz über eine weitere Graphenstruktur, die ihr System über die abgelaufenen Phasen anpasst. Damit soll der Aspekt von guter Modularität gewährleistet werden. Zusätzlich können die einzelnen Funktionen durch den Zugriff auf ihre Zwischenergebnisse profitieren und diese gegebenenfalls direkt modifizieren. Die verwendeten Einzelanalysen müssen korrekt und monoton sein, um ein Ergebnis zu ermöglichen. Des Weiteren müssen ihre Ergebnisse als Graphentransformation darstellbar sein.

Dieses System findet - in der Praxis - Anwendung im Whirlwind- und Vortexcompiler.

4.1 Ablauf

Um mit Hilfe dieser Technik verschiedene Datenflussanalysen zu kombinieren, muss zu allererst eine exakte Spezifikation der einzelnen Analysen vorliegen. Die Vorgehensweise ist sehr modular aufgebaut und man kann sie, bei korrekter Erstellung der erwünschten Analysen, sehr gut erweitern.

Zuerst erläutere ich die Grundidee des Ansatzes. Neben dem normalen Programmflussgraphen wird ein weiterer Graph aufgebaut. Dieser *Replacement Graph* dient der Kommunikation der Analysen und enthält die momentan erwünschten Veränderungen. Eine wichtige Einschränkung ist, dass es nur einen einzigen *Replacement Graph* gibt. Das bedeutet, dass das System nach der Analyse des Grundelements nur eine einzelne Ersetzung vornehmen kann. Schlagen mehrere Analysen verschiedene Ersetzungen vor, müssen diese durch eine *PICK-Funktion* betrachtet und ausgewählt werden. Leider haben Lerner, Grove und Chambers [10] keine genauen oder heuristischen Angaben darüber gemacht, wie eine solche Funktion im einzelnen aussieht.

Mit Hilfe der Funktionen „Propagate“ und „Replace“ wird festgelegt, wann eine Information weitergegeben wird und wann die Konstruktion oder Modifizierung des *Replacement Graph* stattfindet. Der Graph selbst wird auch nur im Falle der „Replace“-Funktion aufgerufen und rekursiv durchgelaufen. Die zugehörigen Elemente speichern die anderen Informationen.

Betrachten wir den Aufbau der Analysen für Konstanten Propagierung und Conditional Branches.

Konstanten Propagierung:

1. k is constant and $x := k ==$
Propagate ($x \rightarrow k$)
2. $(x := y \text{ op } z) ==$
if ($t := y \text{ op } z$ is constant)
then
Replace ($x := t$)
else
Propagate ($x \rightarrow t$)

Die beschriebene Verhaltensweise bedeutet, dass bei dem Vorkommen des ersten Statements der Teil hinter dem doppelten Gleichheitszeichen ausgeführt wird. Plausibilisiert zeigt der unter 4. beschriebene Algorithmus, dass bei einer Zuweisung der Wert für die gegebene Variable abgespeichert wird.

Enthält die Zuweisung eine Operation, dann kommt die zweite Anweisung zum Tragen. Dabei kann es sich auch um einen Variablenwert innerhalb der Zuweisung handeln. Ist der zugewiesene Wert konstant, kann der *Replacement Graph* mit diesen Werten entstehen bzw. modifiziert werden.

Es ist wichtig, dass die Zusammenfassung der Informationen des Verbunds bei Replace- und Propagate-Aktionen zur Verfügung steht. Diese Informationen werden in Maps gespeichert und entsprechend weitergereicht. Auch während der Überprüfung der Bedingung muss man diese Informationen erreichen können. Die Weitergabe der alten Informationen nehmen wir implizit an und erwähnen sie in den Analysebeschreibungen nicht weiter. Dabei findet sie bei jeder im Graphen weiterführenden Aktion statt.

Conditional Branches:

- ```
(if b then goto L1
else goto L2 end) ==
 if (b ist konstant) then
 if (b = true) then
 Replace (Goto L1)
 else
 Replace (Goto L2)
 else
 Propagate()
```

Die Beschreibung zur Auflösung von bedingten Sprüngen nimmt unerreichbaren Quellcode aus der Darstellung des *Replacement Graphs* heraus. Dabei muss der Wert innerhalb der Abfrage konstant sein.

Die Übernahme des *Replacement Graph* für das endgültige Programm darf nur stattfinden, wenn das System einen zutreffenden Fixpunkt erreicht hat. Das bedeutet, dass der *Replacement Graph* sich nicht mehr verändern oder gar unzutreffend sein darf. Um diesen Punkt zu erreichen, müssen

alle einzelnen Datenfluss-Analysen monotone Eigenschaften besitzen. Wenn diese Tatsache nicht gegeben ist, kann der Algorithmus den korrekten Ablauf nicht garantieren. Auch das gesamte System sollte einen monotonen Funktionsverlauf besitzen, da diese Eigenschaft durch das Zusammenspiel der Analysen nicht garantiert werden kann. Die Darstellung zwei möglicher Herangehensweisen an dieses Problem folgt in einem späteren Abschnitt.

Bei der Kombination können die Analysen gegenseitig von ihren Informationen profitieren. Betrachten wir das verwendete Beispiel, um diese Tatsache zu demonstrieren:

```
X := 10;
while (...) {
 if (x == 10) {
 DoSomething();
 } else {
 DoSomethingElse();
 X := X + 1;
 }
}
Y := X;
```

In der ersten Zeile wird der Wert von X weiterpropagiert und liegt danach innerhalb der Schleife vor. Damit kann die Analyse Conditional Branch feststellen, dass nur der erste Sprung ausgeführt wird und der *Replacement Graph* wird mit einer Ersetzung des if-Statements durch eine Goto-Anweisung zur DoSomething-Funktion in der Schleife erstellt. Beim weiteren Verlauf des Systems wird zuletzt der Wert von Y auf 10 festgelegt, da die Information über X durch die Schleife hindurch propagiert wird.

Bei einem erneuten Durchlauf der Schleife kann man feststellen, dass ein Fixpunkt erreicht wurde und damit kann das System den *Replacement Graph* übernehmen.

Durch das Einfügen einer Dekrementierung des X-Wertes nach der Ausführung der DoSomething-Funktion könnte man nur den Wert  $\top$  weiterpropagieren, da die Kombination aus 9 und 10 wie oben beschrieben keine genaue Aussage zulässt. Dabei könnte man keine Vorhersage für das Verhalten während eines weiteren Durchlaufs an der if-Bedingung treffen. Damit müsste der *Replacement Graph* verworfen werden, bevor ein Fixpunkt erreicht wurde.

Weitere Schleifendurchläufe und Neuanalysierungen sind also für die Korrektheit des Verfahrens wichtig. Verzichtet das System auf diese Durchgänge, könnte ein falsches Ergebnis stehenbleiben und sich festsetzen.

Das Ergebnis der gemeinsamen Analyse ist dasselbe wie das Programmfragment beim Ansatz von Wegman und Zadeck [12]. Die Mächtigkeit der beiden Analysetechniken ist im Bezug auf dieses Beispiel also gleich. Nur ist die Modularität der gerade vollführten Analyse deutlich besser gestaltet. Ich ziehe ein zweites umfangreicheres Beispiel heran, um diesen Vorteil näher zu erläutern.

## 4.2 Beispiel

Dieses Beispiel dient zur Verdeutlichung der Modularität und Effektivität dieser Technik [10]. Dabei wird dieses Beispiel umfangreicher gestaltet und im Bereich der Objektorientierung stattfinden. Zu den zwei im System integrierten Analysen wird noch die Klassenanalyse und das Inlining hinzugefügt.

### Klassenanalyse:

1.  $(x := \text{new } C) == \text{Propagate } x \rightarrow C$
2.  $(x := \text{send } y.\text{ID}(\dots)) ==$   
`let (methods = combine  
method_lookup(c, ID))  
if (method = {F}) then  
Replace(x := F(y, z1, ..., zn))  
else  
Propagate(x -> $ \top $)`
3.  $(x := y \text{ instanceof } C) ==$   
`if y $ \subset $ subclasses(C)  
then  
Replace x := true  
else  
if y not in subclasses(C) then  
REPLACE x := false  
else  
PROPAGATE x -> Bool`

Die Klassenanalyse deckt die Bestimmung von neuen Objekten, virtuelle Funktionsaufrufe und Instanceof-Tests ab. Dabei geben die beiden verwendeten Funktionen method\_lookup und subclasses die in Wahrheit aufgerufene Funktion beziehungsweise die zugehörigen Unterklassen der überprüften Klasse zurück.

Der Ansatz benötigt ebenfalls zwei neue Funktionen für das Inlining. Einmal wird should\_inline verwendet. Diese bestimmt eine Heuristik, die entscheidet, ob eine Funktion aufgelöst werden soll. Für eine solche Entscheidungen können nach Wegman und Zadeck [12] generell drei Strategien zur Implementierung ihre Anwendung finden:

**Compiler Direktiven:** Der Programmierer sucht sich selbst die Funktionen aus.

**Statische Analyse:** Der Compiler sucht sich die Funktionen anhand einiger Regeln aus. Zum Beispiel nimmt er nur Funktionen, die an den Blättern eines Aufrufgraphen stehen.

**Leistungsmessungen:** Nach einigen Testläufen des Programms entscheidet der Compiler, auf welche Prozeduren er einen besonderen Fokus legen soll. Diese Möglichkeit ist schwierig umzusetzen, da ein Ergebnis erst nach einigen Compilierungen und mit ordentlichen Testdaten entstehen kann.

Die zweite Funktion ist `subst_formals`. Die Parameter und Return-Statements der Funktion werden entsprechend angepasst, so dass eine Ersetzung des Codestücks stattfinden kann.

### Inlining

```
x := f(y1, ...yn) ==
if should_inline(f) then
 let G=body(f) in
 let G' =
 subst_formals(G, x, y1, ...yn)
 in
 Replace (G')
else
 Propagate()
```

Für das Beispiel benötigen wir die zugrunde gelegten Klassenbeschreibung. Drei Klassen werden im folgenden definiert:

```
class A {
 function foo():A { return new A;}
}
class C extends A {
 function foo():A {return self;}
}
class D extends A {}
```

Die Kombination der vier obigen Einzelanalysen durchläuft dieses Programmfragment:

```
decl x:A;
x := new C;
while (...) {
 decl b: Bool
 b := x instanceof C;
 if (b) {
 x := send x.foo();
 } else {
 x := new D;
 }
}
```

Wenn das System die Schleife betritt, ist der Analyse bekannt, welcher Klasse X in diesem Durchlauf bis zu einer Neuweisung angehört. Die Analyse läuft weiter bis die Konstanten Propagierung den Inhalt von b mit Hilfe des Wissens der Klassenanalyse ableiten kann. Dabei setzt man b direkt auf „wahr“. Da die Bedingung konstant ist, markiert die Analyse des if-Statement den else-Zweig als toten Code. Im *Replacement Graph* wird ein Sprung zum korrekten weiteren Verlauf eingefügt. Man durchläuft zwar noch den anderen Zweig, aber es wird durch den Inhalt des  $\perp$  im Verbund eine leere Ersetzung im *Replacement Graph* vorgenommen. Die Werte im Verbund sind im benutzten

Zweig weitergereicht worden. Dort kann nun das Inlining eingesetzt werden. Mit Hilfe der vorliegenden Klasseninformation kann die aufgerufene Funktion bestimmt werden. Schlägt die `should_inline` Funktion eine Änderung vor, wird diese im Graphen eingefügt. Das Return-Statement bekommt über die Funktion `subst_formal` seinen eigentlichen Inhalt. Damit wird anstelle des Aufrufs X auf X gesetzt.

Die Analyse stellt in einem weiteren Durchgang fest, dass das Gesamtsystem einen Fixpunkt erreicht hat. Denn es liegt keine weitere Änderung der gespeicherten Programminformation vor. Bei diesem erneuten Durchlauf des Programmstücks wird die Analyse schlussendlich den erschaffenen Graphen übernehmen und somit den Quellcode in folgender Form verkürzen:

```
decl x:A;
x := new C;
while (...) {
 decl b: Bool;
 b := true;
 x := x;
}
```

Sequentiell durchgeführte elementare Analysen hätten dieses Ergebnis nicht erzielen können. Erst die Kombination mit integriertem Informationsaustausch ermöglicht es, ein solches Ergebnis zu erzielen. Ein einzelner, kombinierter und angepasster Algorithmus, der diesen vier elementaren Analysen entspricht, besitzt eine deutlich höheren Erstellungsaufwand. Es wären komplexere Überlegungen notwendig, um die vier Analysen zusammenzufassen. Zusätzlich wäre die Adaptionfähigkeit eines solchen Algorithmus deutlich geringer. Durch dieses Beispiel konnte gezeigt werden, dass beim verwendeten Ansatz eine sehr gute Modularität gegeben ist. Den Algorithmus von Wegman und Zadeck [12] mit entsprechenden Elementen zu versehen, würde einen unüberschaubaren Aufwand erfordern. Zusätzlich könnte man für eine entsprechende Laufzeit keine weitere Garantie geben.

Beim vorgestellten Ansatz lassen sich weitere Elemente zwar nur hinzufügen, wenn das Analyseergebnis als Graphentransformation darstellbar ist. Dennoch besitzt er eine Modularität, die mit algorithmisch kombinierten Analysen nicht zu erreichen ist.

## 4.3 Theorie

Durch die Basis der Transformationen auf den Graphen können nur Analysen eingesetzt werden, deren Ergebnisse als Graphtransformationen darstellbar sind. Eine Einbindung anderer Untersuchungstechniken ist nicht möglich. Damit entfällt zum Beispiel das *Instruction Scheduling* und *Loop-Invariant Code Motion*. Diese Techniken müssten in

| Zeilenzahl                   | kombiniert | L/G/C | sequentiell/iterativ | sequentiell |
|------------------------------|------------|-------|----------------------|-------------|
| queens (50) Laufzeit         | 1.00       | 1.02  | 1.25                 | 13.14       |
| Compilezeit                  | 1.00       | 1.17  | 6.17                 | 0.84        |
| life (80) Laufzeit           | 1.00       | 1.00  | 1.09                 | 7.39        |
| Compilezeit                  | 1.00       | 1.17  | 5.72                 | 0.83        |
| msort (110) Laufzeit         | 1.00       | 0.99  | 1.01                 | 6.28        |
| Compilezeit                  | 1.00       | 1.11  | 6.04                 | 0.20        |
| fft (150) Laufzeit           | 1.00       | 1.00  | 0.98                 | 3.00        |
| Compilezeit                  | 1.00       | 1.00  | 6.06                 | 0.71        |
| richards (400) Laufzeit      | 1.00       | 1.00  | 1.07                 | 13.66       |
| Compilezeit                  | 1.00       | 1.18  | 6.52                 | 1.03        |
| deltablue (650) Laufzeit     | 1.00       | 1.03  | 0.94                 | 12.89       |
| Compilezeit                  | 1.00       | 1.20  | 6.54                 | 0.66        |
| instr-shed (2400) Laufzeit   | 1.00       | 1.00  | 1.01                 | 3.78        |
| Compilezeit                  | 1.00       | 1.18  | 5.80                 | 1.02        |
| typechecker (20000) Laufzeit | 1.00       | 1.01  | 1.01                 | 5.30        |
| Compilezeit                  | 1.00       | 1.18  | 6.55                 | 0.94        |
| new-tc (23500) Laufzeit      | 1.00       | 1.05  | 1.03                 | 4.37        |
| Compilezeit                  | 1.00       | 1.17  | 6.09                 | 1.16        |
| compiler (50000) Laufzeit    | 1.00       | 1.02  | 1.00                 | 4.05        |
| Compilezeit                  | 1.00       | 1.15  | 7.46                 | 1.22        |

**Abbildung 1. Laufzeit und Nutzung der Experimente von Lerner, Chambers und Grove [10]**

einem weiteren Durchlauf, nach dem eigentlichen Algorithmus, nochmals durchgeführt werden. Ich weise dabei wieder auf das entstehende Phase Ordering-Problem hin, das dann natürlich auch bei dieser Sequenz auftritt.

Eine weitere Restriktion ist in der Terminierung dieses Vorgehens zu sehen. Im Zusammenspiel der einzelnen Analysen könnte es zu Endlosschleifen kommen, wenn die erste Analyse eine Änderung vornimmt und die zweite diese Änderung wieder auflöst. Man kann diesem Problem nicht mit der Monotonität der einzelnen Analysen beikommen. Es ist für die Gesamtanalyse leider nicht hilfreich, dass die einzelnen Analysen terminieren oder monotone Eigenschaften besitzen. Dieses Problem wird - in gewisser Hinsicht - mit dem sequentiellen Vorgehen geteilt, da man die gesamte zusammengesetzte Analysefunktion betrachten muss.

Die Begrenzung der maximalen Iterationenzahl kann diesem Problem in der Praxis entgegen wirken. Bei einer zweiten Lösungsmöglichkeit könnte man solche Transformation erkennen und sie grundsätzlich umgehen. Dabei wird versucht mit jeder Transformation eine Vereinfachung des Graphen zu bewirken. Das heißt, entweder wird ein Knoten komplett aus dem Graphen gelöscht oder eine komplexe Knotenstruktur wird durch eine deutlich einfachere Struktur ersetzt. Die gewählte Realisierung überlassen die Autoren der direkten Implementierung und geben hier keinen direkten Lösungsweg vor.

Da eine praxisnahe Modularität bei diesem Ansatz im Vordergrund steht, sollte darauf hingewiesen werden, dass

nur eine Analyserichtung zugelassen ist. Das bedeutet, man kann keine Rückwärtsanalyse mit einer Vorwärtsanalyse verbinden. Die Technik unterstützt gleichzeitig nur eine Richtung.

Die Zuverlässigkeit (Soundness) des Systems ist abhängig von den Eigenschaften der einzelnen Analysen. Sind die einzelnen Phasen zuverlässig, ist auch die zusammengesetzte Analyse entsprechend zu betrachten. Ein Beweis dafür wird im zugehörigen technischen Bericht gegeben [9].

#### 4.4 Laufzeit

Das Laufzeitverhalten dieser Technik wurde experimentell ausgetestet. Eine exakt theoretisch fundierte Geschwindigkeit wurde nicht angegeben. Dies liegt sicherlich an dem gegebenen Terminierungsproblemen und der Abhängigkeit zu den verwendeten Analysetechniken.

Um die von den Erstellern gezogenen Schlüsse besser erläutern zu können, stelle ich die Ergebnisse Lerner, Grove und Chambers [10] Experimente mit dem Vortex-Compiler vor. Dabei kann nochmals auf einen Vergleich mit den anderen Kombinationstechniken eingegangen werden.

Die Zahlen wurden mit dem Vortex-Compiler erstellt und es wurden verschiedene Analysen verwendet. Dabei bedeutet die Zeile „Laufzeit“ die gesamte Laufzeit des erstellten Programms, dagegen bedeutet die zweite Zeile die benötigte Compilezeit des Programms.

Die Zahlen sind Faktoren bei der Geschwindigkeitsmessung der Tests. Dabei dient die von Hand zusammengesetz-

te Analyse als Fundament der Tests und ist auf den Wert eins normalisiert. Die Tests haben folgende Analysetechniken verwendet: Klassenanalyse, Splitting, Inlining, Konstanten Propagierung, Common Sub-Expression Elimination, Entfernung von redundanten stores und loads und Symbolic Assertion Propagation.

Die verwendeten kombinierten Algorithmen sind leider nicht mit angegeben und man kann sich dabei nur auf die Erfahrung von Lerner, Grove und Chambers verlassen.

Man sieht, dass der kombinierte Algorithmus in allen Bereichen am Besten abschneidet. Diese Tatsache ist nicht weiter verwunderlich, da diese Technik verschiedene Lösungen problemspezifisch erstellt. Leider ist der in den Experimenten verwendete Algorithmus nicht angegeben und reduziert damit auch die Aussagekraft der Tabelle. Dennoch können auch über den Vergleich mit den restlichen Ergebnissen gute Rückschlüsse gezogen werden.

Die Laufzeit des Ansatzes von Lerner, Grove und Chambers [10] ist mit dem des Gesamtalgorithmus gleichwertig. Allerdings steigt die Compilezeit im Bereich von 20%. Im Gegensatz zu diesen Techniken ist die sequentielle Ausführung der elementaren Datenflussanalysen deutlich ineffizienter. Bei der Verwendung von mehreren Iterationen werden zwar ordentliche Laufzeitergebnisse erreicht, jedoch ist die benötigte Compilezeit um den Faktor 5 erhöht. Ohne die Iterationen in den Sequenzen der einzelnen Analysen sind die Ergebnisse zur Laufzeit leider deutlich schlechter.

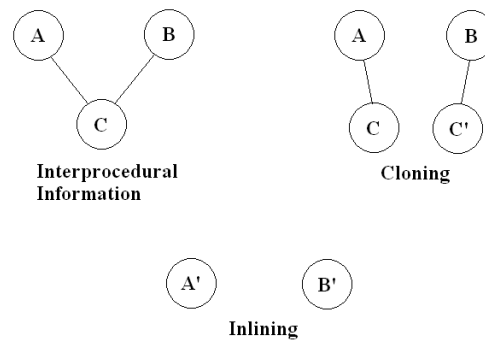
## 5 Interprozedural

Man kann die drei genannten Analysetechniken natürlich auch in einer interprozeduralen Variante benutzen. Hier muss man sich nun zwischen verschiedenen Vorgehensweisen entscheiden. Es gibt drei Wege dieses Problem im allgemeinen für solche Analysen anzugehen:

- Inlining
- Cloning
- Global Optimization mit Interprozedural Informations

In Abbildung 2 [4] sieht man die grafische Erklärung der einzelnen Wege. Dabei stellen A, B und C Prozeduraufrufe dar. Im ursprünglichen Programm rufen die Prozeduren A und B die Prozedur C auf. Der weitere Programmverlauf wird von C bestimmt.

Damit zeigt sich, dass die allgemeine Optimierung versucht den Aufrufgraphen beizubehalten. Dabei wird angestrebt entsprechend aller allgemein gefundener und übergebener Informationen die Prozedur C entsprechend zu modifizieren. Dagegen wird der Graph vom Cloning und vom Inlining entsprechend erweitert, um entsprechende Analysen durchzuführen. Das Cloning teilt die vorhandene Prozedur



**Abbildung 2. Techniken für interprozedurale Erweiterung**

C in zwei getrennt aufgerufene Unterprozeduren auf, um eine bessere Optimierung oder Analyse zu ermöglichen. Das Inlining löst die Unterprozeduraufrufe komplett auf und integriert die Inhalte der Funktionen C in die jeweils aufgerufene Prozedur A und B.

Diese drei Techniken bilden bei M.W.Hall [4] eine Reihenfolge in ihren benötigten Kosten und ihrer Effizienz.

Inlining ist die entsprechend teuerste der drei Methoden, aber es bietet auch die bestmöglichen Resultate, da zu den individuellen Optimierungsmöglichkeiten noch die Einsparung der Kosten für den Prozeduraufruf hinzukommt. Cloning hingegen stellt einen kosten- und effizienzmäßigen Mittelweg dar. Noch günstiger - aber auch am eingeschränktesten in den Möglichkeiten - ist die Optimierung mit Hilfe von interprozeduralen Informationen. Das heißt zum Beispiel, dass im zugehörigen SSA-Graphen - nach möglicherweise einem zusätzlichen Compilerdurchlauf - Informationen über die Übergabeparameter an der Aufrufstelle erstellt wurden. Die SSA-Form kann diese neuen Eigenschaften festhalten.

Eine Begrenzung der Ersetzungstiefe sollte allein wegen der Möglichkeit direkter oder indirekter rekursiver Aufrufe gegeben sein. Trotz dieser Einschränkung würde der Umfang einer gesamten Ersetzung zu einer Explosion in der Programmgröße führen, deswegen sollte das Inlining und Cloning nicht alle Prozeduren ersetzen. Entsprechend der gewählten Strategie müsste eine Implementierung gewählt werden, die den Gesamtumfang stärker begrenzt.

Wegman und Zadeck wählten zunächst für den Sparse Conditional Constant-Algorithmus ein Inlining. Dieses soll natürlich einer zugehörigen Beschränkung unterliegen. Sie schlagen auch einen weiteren Algorithmus vor, der mit Interprozedurale Informationen eine kosteneffizientere aber auch schwächere Lösung ermöglicht.

Zusätzlich kann es bei allen drei Implementierungen wieder zu Permutationsproblemen in der Abfolgereihenfolge der Analyse kommen. Die interprozeduralen Änderungen sollten daher in das Gesamtkonzept der Optimierung eingebunden werden.

## 6. Fazit

Die Thematik der Kombination von Datenfluss-Analysen stand im Zentrum dieser Ausarbeitung. Dabei wurden drei Techniken vorgestellt mit denen die Kombination einzelner elementarer Datenfluss-Analysen möglich ist:

- Sequentielle Kombination
- Kombination durch Verschmelzung zu einer einzelnen Funktion
- Kombination durch ein allgemeines Framework

Wir haben zu allen vorgestellten Vorgehensweisen ein entsprechendes Beispiel herausgegriffen und diskutiert. Diese Beispiele dienen dem leichteren Verständnis der gesamten Problematik und bieten dem Betrachter einen tieferen Einblick in die enthaltenen Schwierigkeiten. Dabei wird der Sparse Conditional Constant-Algorithmus von Wegman und Zadeck [12] sowie das Framework über eine Zwischen-darstellung von Lerner, Grove und Chambers [10] benutzt.

Die Grundvoraussetzungen der Ansätze können unterschiedlich sein, wobei alle eine Verbundsstruktur mit den Informationen über die Variablen enthalten müssen. Man kann diese Struktur je nach geeigneter Implementierung unterschiedlich konstruieren. Der SCC basiert auf der SSA-Form eines gegebenen Programms und kombiniert damit Konstanten Propagierung und Elimination von totem Code. Die allgemeine Kombination über ein System wird mit Hilfe eines Ersetzungsgraphen ermöglicht. Dieses Vorgehen stellt dem Nutzer eine gute Modularität für die Verwendung verschiedener Analysetechniken zur Verfügung. Da sie sich jedoch auf Graphtransformationen gründet, kann zum Beispiel kein Instruction Scheduling betrieben werden, da diese Analyse in dieser Form nicht darstellbar ist.

Die vorgestellten Zahlen sollen die Nutzbarkeit der einzelnen Vorgehensweisen demonstrieren und einen Vergleich bieten. Dabei wurde exemplarisch gezeigt, dass die verschmolzenen Algorithmen die effektivsten Laufzeiten bieten. Der Ansatz über verschiedene Kombinationen steht im Bereich der Laufzeit an zweiter Stelle und eröffnet eine sehr gute Alternative zu dem verschmelzenden Vorgehen.

Sequentielle Kombinationen kämpften in allen Fällen mit der Phase Ordering-Problematik, für die es keine allgemeingültige und schnelle Lösung gibt. Diese Thematik

wurde ebenfalls vorgestellt, um einen besseren Gesamteindruck des Themas zu bekommen und um die Schwierigkeiten einer iterativ, sequentiellen Lösung zu verstehen.

Ob sich mit Hilfe der zwei stärkeren Techniken eine sequentiellen Lösung innerhalb der Analyse ganz vermeiden lässt, kommt auf die Ansprüche der Optimierung an. Beide Möglichkeiten können nicht alle bekannten Optimierungen in einem Durchlauf einbeziehen und es bleibt dem Ersteller überlassen, weitere Techniken in anschließenden Durchläufen zu verwenden. Zum Beispiel muss das Instruction Scheduling noch einbezogen werden, wenn der Ansatz über den *Replacement Graph* verwendet wird.

Schlussendlich wurden noch einige Worte über die Nutzung interprozeduralen Analyseinformationen eingefügt. Der Abschnitt beinhaltet einen kurzen Überblick über die allgemeinen Techniken und Notwendigkeiten solcher Implementierungen. Dabei werden verschiedene Implementierungsmöglichkeiten mit ihren Einzeleigenschaften vorgestellt - das Inlining, Cloning und die Optimierung über die interprozeduralen Informationen. Chambers, Dean und Grove [1] bieten in ihrem technischen Bericht für ihr graphenbasiertes Framework eine interprozedurale Lösung an.

## Literatur

- [1] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical report, University of Washington, 1996.
- [2] C. Chambers und D. Ungar. Iterative type analysis and extended message splitting. Technical report, University of Washington, 1996.
- [3] C. Click und K. D. Cooper. Combining analyses, combining optimization. *ACM Transactions on Programming Languages and Systems*, 1995.
- [4] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.
- [5] L. H. Holley und B. K. Rosen. Qualified dataflow problems. *IEE Trans. Softw. Eng. SE-3*, 3, 1977.
- [6] G. A. Kildall. A unified approach to global program optimization. *Conference Recordings of the First ACM Symposium on Principles of Programming Languages*, 1973.
- [7] P. A. Kulkarni, D. B. Whalley, und G. S. Tyson. Evaluating Heuristic Optimization Phase Order Search Algorithms. *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.
- [8] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, und J. W. Davidson. Exhaustive Optimization Phase Order Space Exploration. *International Symposium on Code Generation and Optimization (CGO'06)*, 2006.
- [9] S. Lerner, D. Grove, und C. Chambers. Composing Dataflow Analyses and Transformation. Technical report, University of Washington, 2001.
- [10] S. Lerner, D. Grove, und C. Chambers. Composing Dataflow Analyses and Transformation. *POPL '02: Conference Record of 29th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2002.

- [11] A. Pioli und M. Hind. Combining interprozedural pointer analysis and conditional constant propagation. Technical report, IBM T.J. Watson Center, 1999.
- [12] M. N. Wegman und F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 1991.