Karlsruhe Institute of Technology

Institut für Programmstrukturen und Datenorganisation (IPD)

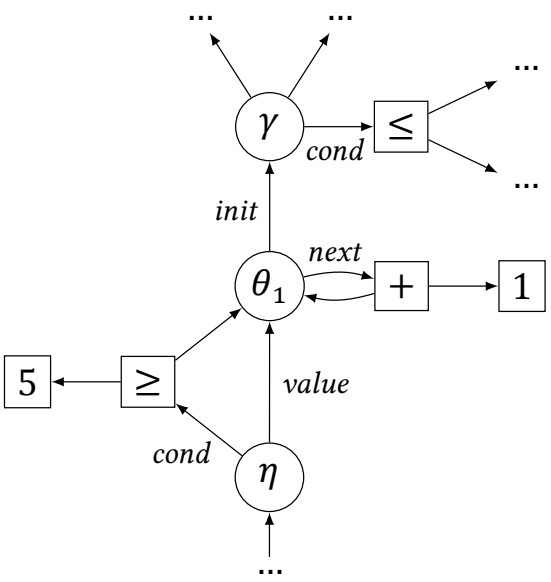
Lehrstuhl Programmierparadigmen
Prof. Dr.-Ing. Gregor Snelting

Eine funktionale und referentiell transparente Zwischensprache für Compiler

Diplomarbeit von

Olaf Liebe

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter: Dipl.-Inform. Matthias Braun

Bearbeitungszeit: 30. September 2010 – 30. Juni 2011

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Ort, Datum Unterschrift

Zusammenfassung

Traditionell orientieren sich Compiler-Zwischensprachen an dem imperativen Aufbau, der bis heute den größten Teil der Programmiersprachen prägt. Hierbei wird der Zustand des Programmes durch eine Reihe von streng geordneten Instruktionen verändert.

Diese Darstellung enthält durch die strenge Ordnung viele Informationen, die für die eigentliche Semantik des Programmes irrelevant sind und ein Hindernis bei Optimierungen darstellen können. Es besteht daher ein Trend, den Steuerfluss zunehmend implizit und die Datenabhängigkeiten zunehmend explizit darzustellen. Auf diese Weise lässt sich die Semantik des Programmes von einer festen Berechnungsstrategie trennen und der Compiler gewinnt die Freiheit, eine eigene Strategie zu wählen. Damit einher geht ein Paradigmenwechsel zu einer funktionalen Sichtweise.

Diese Arbeit stellt eine neue Darstellung namens vFirm vor, welche auf den Firm-, VSDG- und PEG-Darstellungen basiert und sich diesem Trend anschließt. Um die Darstellung praktisch nutzbar zu machen, werden Algorithmen zum Auf- und Abbau von vFirm aus der Firm-Darstellung heraus vorgestellt. Dies ermöglicht die Verwendung von vFirm innerhalb des Firm-Frameworks und damit die Nutzung der vorhandenen Front- und Backends. Darüber hinaus eröffnet es neue Möglichkeiten zur Optimierung von Firm-Programmen.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Compiler sind das Handwerkszeug für die praktische Umsetzung der Informatik, denn sie ermöglichen es, Konzepte und Algorithmen in einer höheren Programmiersprache zum Ausdruck zu bringen und automatisch in eine andere – üblicherweise weniger abstrakte – Sprache zu übersetzen. Durch diese Transformation lässt sich auf einfache Weise ausführbarer Maschinencode erzeugen. Entscheidend ist dabei, dass die Transformation ein semantisch *äquivalentes* Programm erzeugt.

Da aber verschiedene Programme dieselbe Semantik aufweisen können, ist klar, dass diese Transformation nie eindeutig sein kann. Der Compiler nutzt diesen Spielraum, um eine – üblicherweise hinsichtlich des Zeit- und Speicheraufwandes – optimierte Darstellung des Programmes zu erzeugen. Daher ist es erstrebenswert, das Programm zunächst in eine Darstellung zu überführen, die ausschließlich die Semantik des Programmes erfasst und alle weiteren für die reine Transformation irrelevanten Informationen verwirft. Durch diese Reduktion auf eine *normalisierte* Darstellung erhält der Compiler weitere Freiheiten bei der Optimierung des Programmes, da er sich nun stärker von der ursprünglichen Darstellung lösen kann.

1.2 Entwicklung

Betrachtet man verschiedene Zwischendarstellungen, so ist ein genereller Trend zu einer stärkeren Normalisierung zu erkennen. Konkret äußert sich dies darin, dass sich der Fokus vom Steuerfluss hin zu den Datenabhängigkeiten verlagert hat [Law07]. Dies führt in letzter Konsequenz zu den VSDG- und PEG-Darstellungen, auf denen diese Arbeit beruht. Ein kurzer Überblick über diese Entwicklung ist im Folgenden gegeben.

1.2.1 Steuerflussgraph

Viele Compiler verwenden einen Steuerflussgraphen (kurz CFG für Control Flow Graph). Hierbei bilden die Instruktionen streng geordnete Befehlslisten, die als Grundblöcke bezeichnet werden. Die Instruktionen eines Grundblockes werden dabei stets gemeinsam ausgewertet. In einem Grundblock ist die letzte Instruktion stets eine Sprunganweisung, welche (außer

beim Verlassen des Programmes) einen anderen Grundblock zum Ziel hat. Damit bilden die Grundblöcke den Steuerflussgraphen. Daten werden über Variablen und explizite Zuweisungen transportiert und sind damit nur schwer zu erfassen. Somit ist der Steuerfluss explizit gegeben, während die Erfassung von Datenabhängigkeiten zusätzliche Analysen erfordert.

1.2.2 Static-Single-Assignment-Form

Eine bedeutende Abstraktion, die diese Analysephasen vereinfacht und teilweise ersetzt, stellt die sogenannte Static-Single-Assignment-Form (SSA) dar [RWZ88; CFRWZ91], bei der Mehrfachzuweisungen von Variablen nicht möglich sind. Dadurch erhält jede Variable eine eindeutige Definition. Da Variablen ihren dynamischen Charakter verlieren, spricht man auch von *Werten*. Sogenannte ϕ-Instruktionen selektieren dabei abhängig vom Steuerfluss einen von mehreren Werten.

1.2.3 Firm

Die am IPD entwickelte Zwischensprache Firm [Lin02; TLB99] stellt Programme in SSA-Form als Graphen dar. Instruktionen werden hierbei durch Knoten dargestellt, Kanten zwischen den Knoten erlauben die Verwendung der Ergebnisse durch andere Knoten.

Dies wird erst durch die SSA-Form praktikabel, welche eine statische Verknüpfung der Knoten erlaubt. Bei echten Variablen müssten sich die Verbindungen im Graphen zur Laufzeit ändern, so dass ein statischer Graph das Programm nicht erfassen könnte.

Da die ϕ-Knoten jedoch weiterhin einen Steuerfluss benötigen, müssen die Knoten weiterhin in Grundblöcken organisiert werden. Effektiv wurden also die starren Befehlslisten der Grundblöcke durch Graphstrukturen mit einer Halbordnung ersetzt und in den Steuerflussgraphen eingebettet. Es verbleiben aber immer noch explizite Steuerfluss-Strukturen.

1.2.4 VSDG/PEG

Bei den VSDG- und PEG-Darstellungen (Value State Dependency Graph bzw. Program Expression Graph) [JM03; Law07; TSTL09] entfällt nun auch der Steuerflussgraph. Hierdurch ist es möglich, das Programm lediglich durch Datenabhängigkeiten zu definieren und die Unterteilung in Grundblöcke entfällt.

Da der Steuerfluss entfällt, ist eine Werteselektion mittels ϕ-Knoten nicht mehr möglich. An diese Stelle treten sogenannte Gating-Funktionen, die Werte anhand zusätzlicher Bedingungen selektieren. Der Graph wird dadurch weitaus stärker als bisher von einer festen Auswertungsstrategie gelöst.

Beim Erzeugen des ausführbaren Programmes kann der Compiler die hierdurch gewonnene Freiheit ausnutzen, um eine eigene Auswertungsstrategie zu bestimmen.

1.3 Aufbau und Umfang der Arbeit

Kapitel 2 stellt zunächst einige Konventionen und Notationen vor, die im Rahmen der Arbeit verwendet werden. Nachdem diese Basis hergestellt wurde, wird in Kapitel 3 auf die Definition der neuen „vFirm“-Darstellung (*value*-oriented Firm) hingearbeitet, indem die PEG- und VSDG-Darstellungen einer genaueren Betrachtung unterzogen werden. Die eigentliche Definition von vFirm erfolgt in Kapitel 4.

Im Anschluss daran wird in Kapitel 5 die Aufbau-Phase vorgestellt, welche den Aufbau eines vFirm-Programmes aus der Firm-Darstellung heraus beschreibt. Eine entsprechende Abbau-Phase, welche das Program wieder in die Firm-Darstellung zurück transformiert wird in Kapitel 6 vorgestellt. Diese ermöglicht auch die Erzeugung von Maschinencode aus vFirm.

Abschließend werden in Kapitel 7 die erzielten Ergebnisse vorgestellt. Kapitel 8 fasst noch einmal die wichtigsten Erkenntnisse zusammen und gibt einen Ausblick auf mögliche weitere Entwicklungen.

2 Grundlagen

In diesem Kapitel sollen einige grundlegende Notationen eingeführt werden, die in der gesamten Arbeit verwendet werden. Grundsätzlich orientiert sich die Arbeit an den üblichen Notationen im Rahmen der Mathematik und Informatik, einige Notationen sind jedoch spezifisch für diese Arbeit.

2.1 Notation

Die Null- und Einselemente der booleschen Algebra werden mit false und true bezeichnet. Zusätzlich wird $\mathbb{B} = \{\text{true}, \text{false}\}$ definiert. Die Negation wird wahlweise durch ein vorangestelltes $\neg$ oder einen Strich über der Aussage $\bar{A}$ gekennzeichnet. $\mathbb{B}$

Zuweisungen im Sinne der imperativen Programmierung werden durch einen Pfeil dargestellt. $A \leftarrow 0$ weist somit A den Wert 0 zu[1]. Ein unbekannter Wert wird durch $\perp$ beschrieben. $\leftarrow, \perp$

Für Listen und Arrays wird die Notation $[1, 8, 1, 3, 3]$ verwendet. Angelehnt an die Mengenschreibweise sind auch Beispiele wie $[2x \mid x \in \mathbb{N}, x > 9] = [20, 22, ...]$ möglich. Hierbei wird die „naheliegende“ Reihenfolge angenommen, die für gewöhnlich aus dem Kontext ersichtlich ist. $[...]$

2.2 Graphen

Da wir primär graphbasierte Zwischensprachen betrachten, werden auch hierfür zusätzliche Konventionen eingeführt. Sofern nicht anders angegeben, betrachten wir stets einen Graphen $G = (V, E)$ mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$. Einzelne Knoten werden häufig mit $v, w, ...$ oder $v_1, v_2, ...$ bezeichnet, Kanten meist mit $e, e_1, ...$ Gelegentlich kommen aber auch andere Bezeichnungen vor.

Pfade durch einen Graphen werden in der Regel als Abfolge $(v_1, v_2) \rightarrow (v_2, v_3) \rightarrow ... \rightarrow (v_{n-1}, v_n)$ von Kanten aufgefasst und mit $P, P_1, ...$ bezeichnet. Reine Knotenfolgen sind häufig nicht eindeutig, da Mehrfachkanten zwischen Knoten möglich sind.

[1] Diese Konvention soll eine Verwechslung mit dem Vergleich $A = 0$ vermeiden.

2.2.1 Attribute

Attribut	Bedeutung	Beispiele
v.kind	Art des Knotens.	$\phi, \gamma, \eta, \theta$, +, `start`, ...
v.type	Typ des produzierten Wertes.	$\mathbb{B}, \mathbb{N}, \mathbb{T}$, ...
v.depEdges	Liste der Kanten zu den Abhängigkeiten von v.	$[e_1, ..., e_n]$
v.useEdges	Liste der Kanten zu den Benutzern von v.	$[e_1, ..., e_n]$
v.deps	Liste der Abhängigkeiten von v.	$[v_1, ..., v_n]$
v.users	Liste der Benutzer von v.	$[v_1, ..., v_n]$
e.kind	Art/Bedeutung der Kante.	*next*, *init*, *cond*, ...
e.src	Ursprungsknoten.	v, w, ...
e.dst	Zielknoten.	v, w, ...

Tabelle 2.1: Wichtige Attribute für Knoten v und Kanten e

Knoten und Kanten alleine sind selten ausreichend, um einen bestimmten Algorithmus oder Sachverhalt zu erklären. Aus diesem Grund wird die Notation um sogenannte Attribute erweitert. Diese ermöglichen es, Werte mit bestimmten Knoten oder Kanten zu verknüpfen.

Für solche Attribute wird die Notation v.attr verwendet, wobei v in diesem Fall ein Knoten ist. Das Prinzip lässt sich aber auch auf Kanten und beliebige andere Objekte erweitern. Beispielsweise wird häufig e.dst verwendet, um auf den Zielknoten der Kante e zu verweisen. Weitere häufig verwendete Attribute sind in Tabelle 2.1 aufgelistet.

Zur Formulierung eines Algorithmus ist es häufig praktisch, einen Standardwert für ein Attribut vorzugeben. Dieser gilt dann solange, bis dem Attribut ein anderer Wert zugewiesen wird. Zu diesem Zweck wird die Notation auf Mengen verallgemeinert, so dass sich M.attr auf alle Elemente von M bezieht. Eine Zuweisung M.attr $\leftarrow$ ... setzt daher das Attribut attr für alle Elemente in M. Mit $M = V$ lässt sich damit ein Standardwert für alle Knoten festlegen.

Wenn kein expliziter Standardwert gegeben ist, wird im Allgemeinen davon ausgegangen, dass dieser 0, $\perp$ oder false ist. Dies sollte normalerweise aus dem Kontext ersichtlich sein.

Beispiel

Betrachtet man den Vergleich zweier Ganzzahlen, wie in Abbildung 2.1 dargestellt, so würde man beispielsweise folgende Attribute vorfinden:

$$
\begin{array}{ccc}
v.\text{kind} = \leq & w.\text{kind} = \texttt{const} & x.\text{kind} = \texttt{const} \\
v.\text{type} = \mathbb{B} & w.\text{type} = \mathbb{N} & x.\text{type} = \mathbb{N} \\
v.\text{depEdges} = [e_w, e_x] & w.\text{useEdges} = [e_w] & x.\text{useEdges} = [e_x] \\
v.\text{deps} = [w, x] & w.\text{users} = [v] & x.\text{users} = [v] \\
e_w.\text{src} = v & e_w.\text{dst} = w & e_x.\text{dst} = x \\
e_x.\text{src} = v & e_w.\text{kind} = \textit{left} & e_x.\text{kind} = \textit{right}
\end{array}
$$

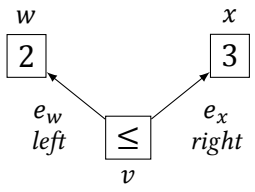


Abbildung 2.1: Attributierter Graph

2.2.2 Knoten als Funktion

Bei der Betrachtung von Zwischendarstellungen kann ein Knoten häufig als Funktion betrachtet werden, die aus den Eingabewerten (in der Regel) genau einen Ausgabewert produziert. Betrachten wir dazu erneut das Beispiel aus Abbildung 2.1.

Der Knoten v repräsentiert den Wert $2 \leq 3$ und lässt sich als Funktion $f(w, x) = \text{value}(w) \leq \text{value}(x)$ auffassen[2]. Anders gesagt: der Knoten repräsentiert das Ergebnis, welches sich durch Anwendung der Funktion f auf die Werte der Abhängigkeiten ergibt. Damit lässt sich die Semantik der Knoten unabhängig von konkreten Werten beschreiben.

Daher werden Knoten meist als Funktion $f(par_1, ..., par_n)$ mit abstrakten Parametern definiert. f ist hierbei meist ein symbolischer Name, in diesem Fall „$\leq$“. Die passende Definition wäre $\leq(left, right) := \text{value}(left) \leq \text{value}(right)$. Ein Beispiel für die Anwendung auf konkrete Werte ist $\leq(2, 3) = 2 \leq 3 = \text{true}$.

Davon abgesehen liefert die Notation mit abstrakten Parameternamen eine Spezifikation des Knotentyps $v.\text{kind} = \leq$, sowie implizit die Definition verschiedener Attribute. So bezeichnen $v.\text{left} = w$ und $v.\text{right} = x$ nun die Knoten an den entsprechenden Eingängen. Die Kanten werden entsprechend mit $v.\text{leftEdge} = e_w$ und $v.\text{rightEdge} = e_x$ bezeichnet. Auch die Art der Kanten leitet sich hieraus ab: $e_w.\text{kind} = left$ und $e_x.\text{kind} = right$.

2.3 Spezielle Darstellungen

Durch einige Analysen lassen sich wichtige Datenstrukturen berechnen, die für spätere Algorithmen von Relevanz sind.

2.3.1 Steuerflussgraphen

Steuerflussgraphen werden im Rahmen dieser Arbeit nie zur Darstellung eines gesamten Programmes verwendet, sondern beschreiben stets nur eine Funktion im Programm. Für die Beschreibung des Steuerflussgraphen lehnen wir uns an die Firm-Darstellung an.

[2]Die value-Funktion bestimmt das Ergebnis eines Knotens und wird später konkretisiert.

Definition 1 (Grundblock). *Ein Grundblock beschreibt eine Menge von Instruktionen. Mit den folgenden Eigenschaften:*

1. *Instruktionen in einem Grundblock werden stets gemeinsam ausgeführt. Wenn eine Instruktion des Grundblockes ausgeführt wird, dann auch alle anderen.*
2. *Ein Grundblock enthält genau eine Sprunginstruktion, welche – außer beim Verlassen des Programmes – einen anderen Grundblock zum Ziel hat.*

`jump`, `cond` `return`

Als Sprunginstruktionen zählen wir hierbei den unbedingten Sprung `jump`, den bedingten Sprung `cond` sowie `return`. Durch sie werden die Grundblöcke zu einem Graphen verknüpft. Dabei gibt es immer einen Start-Block v_s, über den die Funktion betreten wird, sowie einen End-Block v_e welcher die Funktion mit einer `end`-Instruktion verlässt. Der `return`-Sprung hat die Eigenschaft, dass er immer den End-Block zum Ziel hat und einen Rückgabewert entgegennehmen kann.

block nodes

Wir werden in der Regel nur Steuerflussgraphen in der Firm-Darstellung betrachten. Hierbei werden die Instruktionen innerhalb der Grundblöcke auch als Graphen dargestellt, so dass jede Instruktion durch einen Knoten repräsentiert wird (siehe Abschnitt 2.2.2). Der Grundblock, der den Knoten v enthält, wird als v.block bezeichnet. Umgekehrt lässt sich für einen Grundblock w die Menge der enthaltenen Knoten durch w.nodes erfragen.

Da Kanten in Firm Abhängigkeiten beschreiben, sind die Steuerflusskanten der eigentlichen Ausführungsrichtung *entgegen* gerichtet. Beispielsweise ist der End-Block v_e von allen `return`-Knoten abhängig. Nach Abschnitt 2.2.1 bezeichnet v_e.deps diese `return`-Knoten.

preds succs predEdges succEdges

Für den Steuerfluss ist es aber häufig intuitiver, von Vorgänger- und Nachfolgerblöcken zu reden und sich dabei auf die Ausführungsrichtung des Programmes zu beziehen. Aus diesem Grunde werden die Attribute users und useEdges bei Grundblöcken durch preds und predEdges ergänzt, welche die Vorgänger-Blöcke und -Kanten bezeichnen. Für die Nachfolger-Blöcke werden entsprechend succs und succEdges definiert. Es ist jedoch zu beachten, dass die Kanten nach wie vor Abhängigkeiten beschreiben. Die Kanten v_e.predEdges zeigen beispielsweise vom End-Knoten zu den jeweiligen `return`-Knoten.

2.3.2 Baumstrukturen

parent children

Für baumförmige Graphen lässt sich eine Hierarchie aus Vater- und Kinder-Knoten definieren. Wir repräsentieren diese ebenfalls in Form von Attributen. Dabei bezeichnet v.parent den Vater-Knoten von v und v.children die Menge der Kinder-Knoten von v.

Die Wurzel w ist der einzige Knoten im Graphen mit $w.\text{parent} = \bot$. Die Blätter u des Baumes sind die einzigen Knoten mit $u.\text{children} = \emptyset$.

2.3.3 Dominanzbaum

Der Dominanzbaum ist eine Darstellung der für Flussgraphen bekannten Dominanzrelation [AU77]. Sie kann immer dann definiert werden, wenn man einen Graphen $G = (V, E)$ mit einem ausgezeichneten Start-Knoten v_s betrachtet. Wir werden diese Definition später sowohl auf den Steuerflussgraphen als auch auf vFirm beziehen.

Definition 2 (Dominator). *Ein Knoten v dominiert den Knoten w, wenn jeder Pfad vom Start-Knoten v_s nach w über v führt. Man schreibt dann auch: $v \operatorname{dom}^* w$ und bezeichnet v als* Dominator *von w. Gilt außerdem $v \neq w$, so spricht man von* strikter Dominanz *und schreibt $v \operatorname{dom}^+ w$.*

dom^*
dom^+

Die Relation ist reflexiv, transitiv und antisymmetrisch. Sie induziert also eine Halbordnung auf der Menge der Knoten. Da jeder Knoten mehrere Dominatoren besitzen kann, ist die Definition eines eindeutigen Dominators hilfreich:

Definition 3 (Direkter Dominator). *Der eindeutige Knoten v mit:*

$\operatorname{idom}(v)$

$$[v \operatorname{dom}^+ w] \wedge [\forall x \in V : (x \operatorname{dom}^+ w) \rightarrow (x \operatorname{dom}^* v)]$$

heißt direkter Dominator von w und wird auch als $\operatorname{idom}(w)$ *bezeichnet. Dies ist der einzige Knoten $v \neq w$, der keinen anderen Dominator von w dominiert, w selbst aber schon.*

Solange alle Knoten im Graphen vom Start-Knoten aus erreicht werden können, ist der durch idom induzierte Graph ein Baum. Dabei ist v_s die Wurzel des Baumes und zwei Knoten v und w sind genau dann mit einer Kante (v, w) verbunden, wenn $v = \operatorname{idom}(w)$.

Der gesamte Dominanzbaum wird als $D(v_s)$ bezeichnet. Entsprechend wird der Teilbaum unter einem beliebigen Knoten v als $D(v)$ bezeichnet. Er ist genau dann leer, wenn der Knoten keinen anderen Knoten dominiert und damit ein Blatt im Dominanzbaum darstellt.

$D(v)$

Die Dominanzrelation lässt sich durch zwei Vergleiche in konstanter Zeit abfragen, wenn man die Knoten im Dominanzbaum per Tiefensuche nummeriert und an jedem Knoten den jeweils kleinsten und größten Index im Teilbaum vermerkt.

Definition 4 (Nächster gemeinsamer Dominator). *Wir erweitern die Dominanzrelation auf Knotenmengen M. Dabei ist der eindeutige Knoten v mit:*

$\operatorname{dom}(M)$

$$[\forall w \in M : v \operatorname{dom}^* w] \wedge [\forall x \in V : (\forall w \in M : x \operatorname{dom}^* w) \rightarrow (x \operatorname{dom}^* v)]$$

der nächste gemeinsame Dominator von M oder auch $\operatorname{dom}(M)$.

Die Berechnung von $\operatorname{dom}(M)$ kann effektiv durchgeführt werden, indem von einem beliebigen Knoten $v \in M$ aus im Dominanzbaum aufgestiegen wird, bis ein Knoten w mit $M \subseteq D(w)$ gefunden wird. Dann ist $\operatorname{dom}(M) = w$. Für $M = \{v\}$ gilt dabei $\operatorname{dom}(M) = v$.

2.3.4 Schleifenbaum

Da Schleifen verschachtelt sein können, weisen sie eine Hierarchie auf, die sich in einem Schleifenbaum darstellen lässt [Ram02]. Dabei repräsentiert jeder Knoten im Baum eine Schleife, die einen oder mehrere Grundblöcke umfassen kann. Die Knoten des Schleifenbaumes werden üblicherweise mit $L, L_0, L_1, \ldots$ bezeichnet.

depth Der Abstand zur Wurzel oder die *Tiefe* der Schleife wird über das depth-Attribut abgebildet. Der Wurzelknoten L_0 hat dabei die Tiefe $L_0.\text{depth} = 0$ und repräsentiert den gesamten Graphen. Er ist der einzige Knoten, der keine wirkliche Schleife darstellt.

blocks loop Für einen beliebigen Knoten L des Schleifenbaumes ist L.blocks die Menge aller Grundblöcke, welche die hierdurch repräsentierte Schleife umfasst. Dies beinhaltet die Grundblöcke aller Schleifen, die L untergeordnet sind. Umgekehrt ordnet v.loop dem Grundblock v die umschließende Schleife mit der höchsten Tiefe zu.

header Für Schleifen L, die *reduzibel* sind (siehe [HU74]) existiert außerdem ein eindeutiger Schleifenkopf. Dies ist der Grundblock, über den die Schleife betreten wird. Er wird als L.header bezeichnet.

3 Verwandte Arbeiten

PEG und VSDG ziehen wesentliche Konzepte aus der SSA-Darstellung und dem Program Dependence Graph (PDG) [FOW87]. Sie bilden die Basis für die von Ballance, Maccabe und Ottenstein entwickelte Program-Dependence-Web-Darstellung [BMO90] (PDW), welche die Zwischendarstellung erstmals um zusätzliche Gating-Funktionen erweitert.

Diese nehmen den Platz der aus der SSA-Darstellung bekannten ϕ-Knoten ein und ermöglichen anhand zusätzlicher Bedingungen – auch ohne expliziten Steuerfluss – eine Werteselektion. Daher kommen sie auch in den VSDG- und PEG-Darstellungen zum Einsatz. Die um Gating-Funktionen ergänzte PDG-Darstellung wird auch als *Gated-SSA*-Form (GSA) bezeichnet.

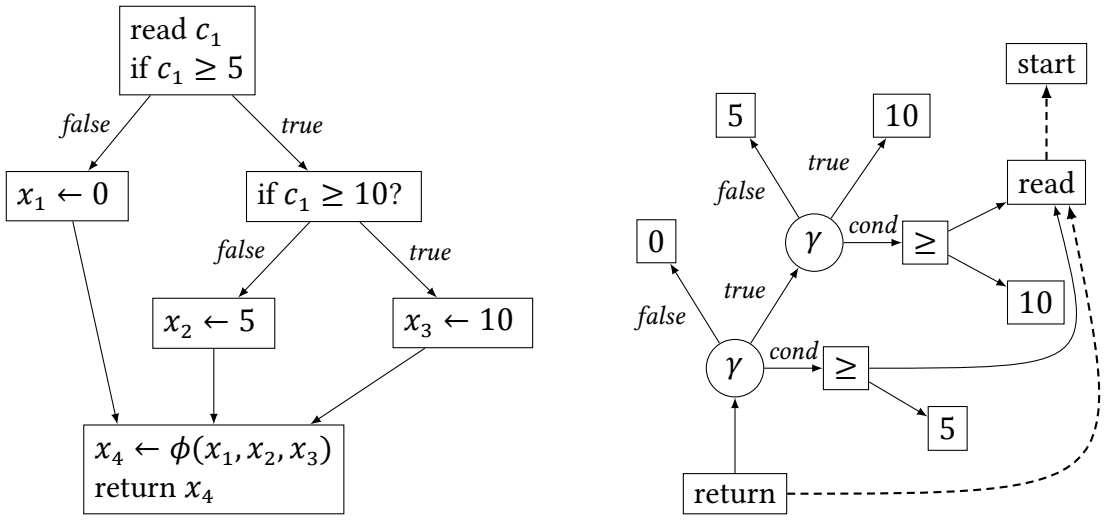
Aufgrund der PDG-Darstellung und dem Anspruch der Zwischendarstellung, auf verschiedene Weisen interpretierbar zu sein, wurden die Steuerflusskanten jedoch nicht vollständig fallen gelassen. Dieser Schritt bleibt dem von Weise et al. entwickelten *Value Dependence Graph* vorbehalten [WCES94], welcher ausschließlich auf Basis von Datenabhängigkeiten definiert ist und die Basis für VSDG- und PEG-Darstellung legt.

3.1 Value State Dependence Graph

Der *Value State Dependence Graph* (VSDG) ist eine von Johnson und Mycroft entwickelte Darstellung auf der Basis des VDG [JM03]. Die zentrale Änderung stellt die Ergänzung um Zustands-Werte und -Kanten dar, durch die der VSDG voneinander unabhängige Operationen mit Nebeneffekten (beispielsweise Lade- und Speicheroperationen) sinnvoll ordnen kann. Eine ausführliche Betrachtung der Sequentialisierung – also der Überführung zurück in einen Steuerflussgraphen – erfolgt in der Dissertation von Lawrence [Law07].

3.2 Program Expression Graph

Der *Program Expression Graph* (PEG) ist eine weitere Darstellung von Tate et al. [TSTL09] mit starken Parallelen zum VSDG und Unterschieden in der Schleifendarstellung. Ein technischer Bericht [TSTL11] beschreibt die Übersetzung zwischen Steuerflussgraph und PEG.



(a) CFG in SSA-Form (b) VSDG/PEG

Abbildung 3.1: Verzweigungen in der VSDG-/PEG-Darstellung

3.3 Gating-Funktionen

3.3.1 Verzweigungen

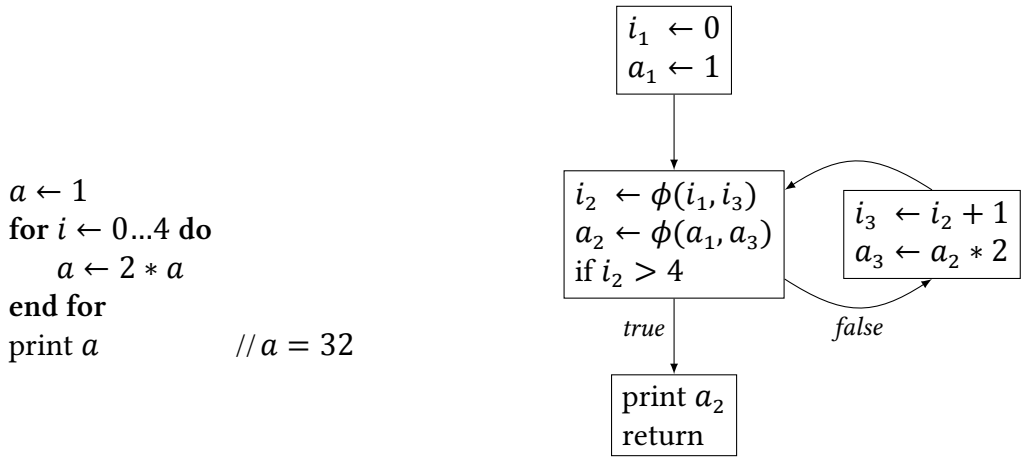
Die in den genannten Darstellungen verwendeten Gating-Funktionen weisen starke Ähnlichkeiten auf und können daher an diesem Punkt gemeinsam vorgestellt werden. Identisch ist in allen Darstellungen die γ-Funktion[1], welche den geeigneten Ersatz für die ϕ-Funktion der SSA-Form bei einfachen (nicht zyklischen) Verzweigungen darstellt. Bei der Vorstellung von vFirm erfolgt eine genauere Definition. Vorerst ist eine textuelle Beschreibung jedoch ausreichend:

Vorläufige Definition 1 (γ-Knoten). *Der Knoten* $\gamma(cond, value_{\text{true}}, value_{\text{false}})$ *wertet die Bedingung cond aus und gibt* $value_{\text{true}}$ *zurück, wenn das Ergebnis der Auswertung* true *ist. Ansonsten wird* $value_{\text{false}}$ *zurückgeliefert.*

Abbildung 3.1 zeigt, wie ein γ-Knoten die ϕ-Funktionen in der SSA-Darstellung ersetzen kann. Die Darstellung als VSDG- und PEG-Graph ist hierbei identisch, da noch keine Schleifen involviert sind. Es ist zu beachten, dass der unstrukturierte Steuerfluss im CFG die Verwendung mehrerer γ-Knoten erfordert, um eine ϕ-Funktion zu ersetzen. Die Bedingungen der γ-Knoten ergeben sich dabei aus den Verzweigungs-Bedingungen des CFG.

Gestrichelt dargestellt sind die Zustandskanten. Ein expliziter `start`-Knoten ist notwendig, um einen ersten Zustand zu erzeugen. Da das Lesen der Eingabe Nebeneffekte hervorruft, benötigt der `read`-Knoten einen Zustand als Eingabe und gibt ihn modifiziert wieder zurück.

[1]Diese wird in der PEG-Darstellung als ϕ-Funktion bezeichnet. Allerdings mit anderer Semantik als die ϕ-Funktion der SSA-Darstellung, weshalb hier stets die Bezeichnung γ-Funktion vorgezogen wird.



(a) Pseudocode (b) CFG

Abbildung 3.2: Schleifen in der SSA-Darstellung

3.3.2 Schleifen

Schleifen können Zyklen im Wertefluss hervorrufen. Dies ist in Abbildung 3.2 zu sehen. Die ϕ-Funktionen sind rekursiv definiert, so dass die Zyklen $i_2 \rightarrow i_3 \rightarrow i_2$ und $a_2 \rightarrow a_3 \rightarrow a_2$ im Wertefluss entstehen. Je nach Darstellung werden diese Fälle unterschiedlich gehandhabt.

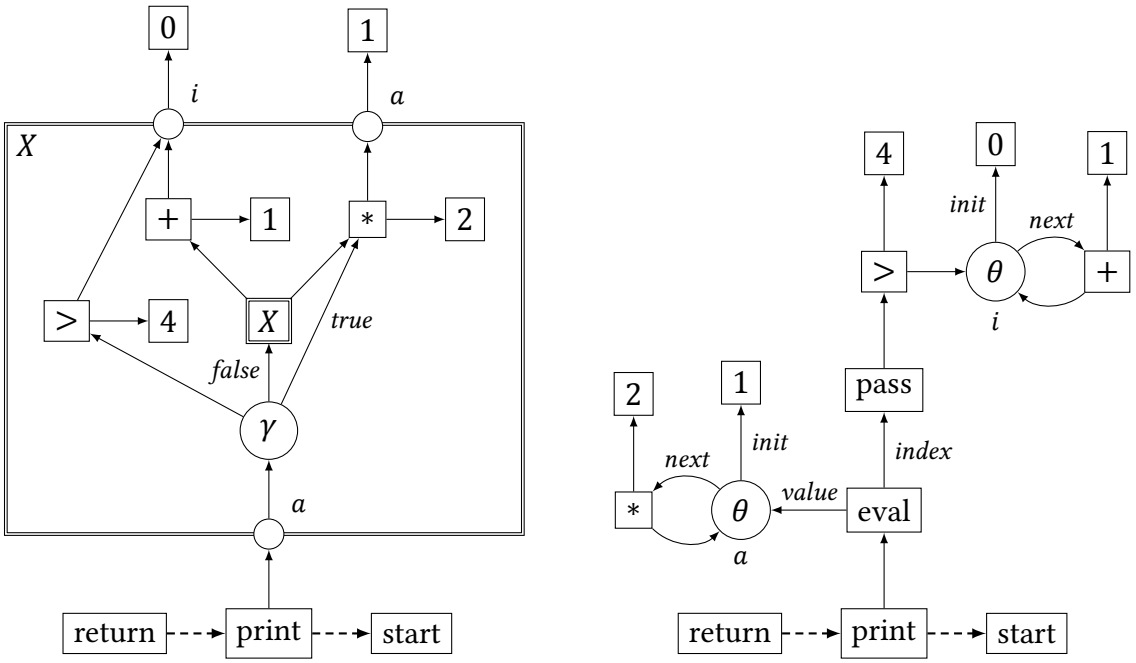
Rekursiver Graph

Diese Variante wird von Lawrence' VSDG-Darstellung [Law07] und von der ursprünglichen VDG-Darstellung [WCES94] verwendet. Hierbei wird ein Teil des Graphen herausgegriffen und rekursiv verwendet.

Der zu Abbildung 3.2 passende Graph in dieser Darstellung ist in Abbildung 3.3a abgebildet. Dabei repräsentiert der Knoten X mit dem doppelten Rand den umschließenden, ebenfalls doppelt gerahmten Graphen. Durch diese Konstruktion ist eine Darstellung ausschließlich mit Hilfe des γ-Knotens möglich. Der Graph bleibt azyklisch, allerdings stellt die Rekursion einen zyklischen Bezug dar.

Zyklischer Graph

Alternativ kann man die Zyklen auch im Wertefluss belassen und den Graphen um weitere Gating-Funktionen ergänzen, welche die wiederholte Auswertung steuern. Dieser Weg wird in der ursprünglichen Definition der VSDG-Darstellung [JM03], sowie in der PEG-Darstellung beschritten [TSTL09]. Im Folgenden wird dies an der PEG-Darstellung vorgestellt.



(a) Rekursive Darstellung (b) Zyklische Darstellung

Abbildung 3.3: Schleifen in der SSA-Darstellung

Der zyklische Wertefluss, wie er in Abbildung 3.2 dargestellt wurde, beginnt mit einem Initialwert (i_1) und durchläuft dann wiederholt dieselben Operationen (i_2, i_3). Dieser Wertefluss wird in der PEG-Darstellung durch den θ-Knoten definiert, der genau eine solche zyklische Auswertung beschreibt.

Vorläufige Definition 2 (θ-Knoten). *Der Knoten $n \coloneqq \theta(init, next)$ beginnt einen Zyklus im Wertefluss. Dabei ist init der Initialwert und next eine auf n verweisende zyklische Definition.*

Von einem deklarativen Standpunkt aus kann man sagen, dass der θ-Knoten eine unendliche Liste von Werten definiert, die sich induktiv durch wiederholtes Auswerten der zyklischen Definition, begonnen mit dem Initialwert ergibt. Am Beispiel des θ-Knotens a (siehe Abbildung 3.3b) erhält man somit:

$$
\begin{aligned}
a_0 &\coloneqq init = 1 && \text{(Initialwert)}\\
a_1 &\coloneqq 2 \cdot a_0 = 2 && \text{(1. Wiederholung)}\\
a_2 &\coloneqq 2 \cdot a_1 = 4 && \text{(2. Wiederholung)}\\
&\ldots\\
a &= [1, 2, 4, 8, 16, 32, \ldots] && \text{(Unendliche Liste)}
\end{aligned}
$$

Auf diese Weise kann man *jede* Operation, die auf einen θ-Knoten zurückgeht, als unendliche Liste betrachten. Beispielsweise wird der Vergleich $x > 4$ auf den θ-Knoten i und damit auf

die unendlich vielen Werte $i = [0, 1, 2, ...]$ angewendet. Man erhält:

$$\begin{aligned} r_0 &\coloneqq (i_0 > 4) = \text{false} \\ r_1 &\coloneqq (i_1 > 4) = \text{false} \\ &... \\ r &= [\text{false}, \text{false}, \text{false}, \text{false}, \text{false}, \text{true}, ...] \end{aligned}$$

Bis zu diesem Punkt beschreibt der θ-Knoten ausschließlich den zyklischen Wertefluss, sagt jedoch noch nichts darüber aus, wann ein Wert extrahiert werden muss, beziehungsweise wann die Auswertung abbricht. Zu diesem Zweck definiert die PEG-Darstellung die `eval`- und `pass`-Knoten.

Definition 5 (`pass`-Knoten). *Der `pass`(value)-Knoten erhält als Argument eine unendliche Liste value von booleschen Werten und liefert den Index des ersten Elementes mit dem Wert* true *zurück.*

Bezogen auf das obige Beispiel wäre also $\text{pass}(r) = 5$. Dies entspricht genau der Anzahl an Iterationen, da $x > 4$ die Abbruchbedingung der Schleife ist. Um also den Wert a nach Abbruch der Schleife zu erhalten, muss mit diesem Index auf die Liste von a zugegriffen werden. Dies ist Aufgabe des `eval`-Knotens:

Definition 6 (`eval`-Knoten). *Der `eval`(value, index)-Knoten extrahiert den Wert mit dem gegebenen Index aus der Liste value und gibt diesen zurück.*

Bezogen auf das Beispiel ist damit:

$$\text{eval}(a, \text{pass}(r)) = \text{eval}([1, 2, 4, 8, 16, 32, ...], 5) = 32$$

Die Verwendung von `eval` im Graphen ist in Abbildung 3.3b dargestellt. Es ist zu beachten, dass bei dieser Darstellung eine einzige Schleife in diverse θ-Knoten zerfallen kann, denn jede Schleife kann mehrere zyklische Werteflüsse enthalten.

3.4 Auswertungssemantik

Die Verwendung von Gating-Funktionen verändert die Auswertungssemantik fundamental. Will man einen CFG interpretieren, so kann man ausgehend vom Startknoten den Graphen von oben nach unten durchlaufen und dabei alle Operationen eines Grundblockes gemeinsam auswerten, um anschließend zum Nachfolge-Block zu springen.

Die Interpretation der VSDG/PEG-Graphen beginnt im Gegensatz dazu beim `return`-Knoten am Ende des Programmes. Danach erfolgt die Auswertung *bei Bedarf*: das heißt, um einen Knoten auszuwerten, müssen zunächst alle seine Abhängigkeiten ausgewertet werden. Für

die Abhängigkeiten wiederholt sich die Strategie, so lange bis man an einem Knoten ohne Abhängigkeiten ankommt.

Die Ausnahme von dieser Regel bilden die Gating-Funktionen, welche eine selektive Auswertung der Abhängigkeiten erlauben. Die Semantik des Graphen definiert sich damit bottom-up, ausgehend von der zwingenden Auswertung des `return`-Knotens.

4 Die vFirm-Darstellung

4.1 Definition

Die vFirm-Zwischendarstellung (*value-oriented* Firm) soll im Folgenden formalisiert und vorgestellt werden. Es gilt zunächst:

Definition 7 (vFirm). *Ein vFirm-Programm ist ein attributierter und gerichteter Graph* $G = (V, E, v_s, v_r)$ *mit Knoten* V*, Kanten* $E \subseteq V \times V$ *sowie* `start`*-Knoten* v_s *und* `return`*-Knoten* v_r*.*

Um im Folgenden die Semantik eines Knoten zu beschreiben, wird die sogenannte value-Funktion verwendet. Sie ordnet einem Knoten sein Resultat zu und wird dann für die hier definierten Knoten genauer spezifiziert.

Definition 8 (value-Funktion). *Für einen Knoten* v *ist* $\text{value}_L(v) \in v.type$ *der durch den Knoten berechnete Wert. Da bei Schleifen je nach Kontext verschiedene Werte produziert werden, gibt L die Indizes der involvierten Schleifen an. Dies wird später noch konkretisiert.*

4.2 Verzweigungen

Damit lässt sich der γ-Knoten formal definieren als:

Definition 9 (γ-Knoten). *Ein Knoten* $v = \gamma(cond, value_{\text{true}}, value_{\text{false}})$ *selektiert abhängig von cond eines der Argumente* $value_{\text{true}}$ *bzw.* $value_{\text{false}}$*. Es gilt:*

$$\text{value}_L(v) = \begin{cases} \text{value}_L(v.value_{\text{true}}), & \textit{falls } \text{value}_L(v.cond) = \text{true}, \\ \text{value}_L(v.value_{\text{false}}), & \textit{falls } \text{value}_L(v.cond) = \text{false}. \end{cases}$$

Entsprechend ist $\text{value}_L(\gamma(\text{true}, +(2,5), 0)) = 7$ oder $\text{value}_L(\gamma(=(8,9), 0, 1)) = 1$, denn es gilt $\text{value}_L(+(2,5)) = 7$ sowie $\text{value}_L(=(8,9)) = \text{false}$.

4.3 Schleifen

Bei der Schleifendarstellung fiel die Entscheidung zugunsten der PEG-Darstellung. Das heißt, wir verwenden für die Schleifen zusätzliche Gating-Funktionen und einen (eingeschränkt) zyklischen Graphen. Dafür waren folgende Gründe ausschlaggebend:

1. Die Integration in das bestehende Firm-System fällt leichter, da die hierarchische Unterteilung in der rekursiven Darstellung entfällt.
2. Die bei der VSDG-Darstellung verwendete rekursive Definition verwendet γ-Knoten, die auf ganzen Werte-Tupeln arbeiten und hält hierdurch die Knoten der Schleife zusammen. Bei der PEG-Darstellung zerfällt die Schleife in θ-Knoten für jeden Ausdruck. Dies erleichtert die Optimierung der individuellen Ausdrücke.
3. Da der zyklische Datenfluss explizit durch Gating-Knoten zum Ausdruck gebracht wird, lassen sich viele Schleifen-Optimierungen leichter ausdrücken [TSTL09]. Beispielsweise lässt sich eine Induktionsvariable wie $*(5, v \coloneqq \theta(0, +(v, 1)))$ einfach in $v \coloneqq \theta(0, +(v, 5))$ transformieren, um die Multiplikation zu eliminieren.

Im Vergleich zur PEG-Darstellung und angelehnt an die Gated-SSA-Darstellung [BMO90] wurden die `eval`- und `pass`-Knoten zu einem einzigen η-Knoten zusammengefasst. Dies ist sinnvoll, da die Knoten in der PEG-Darstellung ohnehin nur gemeinsam auftreten.

Davon abgesehen werfen getrennte Knoten hier verschiedene Probleme auf. Die Verwendung des `pass`-Knotens ohne korrespondierendes `eval` wäre zwar denkbar, ist aber praktisch nicht erforderlich. Der bei der PEG-Darstellung angegebene Abbau [TSTL11] behandelt diesen Fall erst gar nicht, so dass ein solches Programm als inkorrekt betrachtet werden kann. Davon abgesehen wäre unklar, welchen konkreten Rückgabetyp (32-Bit Integer, …) `pass` haben müsste, denn das Ergebnis kann potentiell beliebig groß sein. Der η-Knoten vermeidet diese Probleme.

Definition 10 (η-Knoten). *Ein Knoten $v = \eta(cond, value)$ selektiert abhängig von cond einen der vielen Werte, die value annehmen kann:*

$$j = \min\{i \mid i \in \mathbb{N}_0, \mathrm{value}_L(v.cond)_i = \mathrm{true}\} \cup \{\infty\}$$
$$\mathrm{value}_L(v) = \begin{cases} \mathrm{value}_L(v.value)_j, & \textit{falls } j < \infty, \\ \bot, & \textit{sonst.} \end{cases}$$

Drückt man die Werte durch unendlich lange Listen aus, so erhält man zum Beispiel:

$$\mathrm{value}_L(\eta([\mathrm{false}, \mathrm{false}, \mathrm{false}, \mathrm{true}, \ldots], [2, 4, 6, 8, \ldots])) = 8$$
$$\mathrm{value}_L(\eta([\mathrm{false}, \ldots], [2, 4, 6, 8, \ldots])) = \bot$$

Der zweite Fall repräsentiert dabei eine Endlosschleife. Entsprechend liefert der η-Knoten das undefinierte Ergebnis $\bot$.

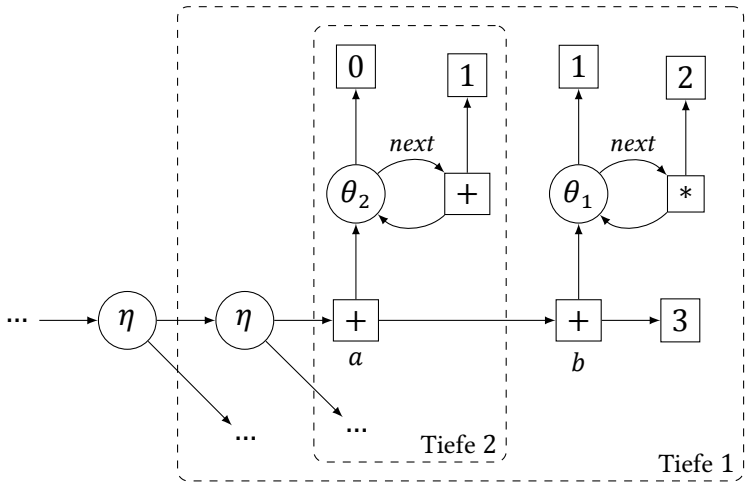


Abbildung 4.1: Verschachtelte Schleifen

4.3.1 Schleifentiefe

Für die Definition des θ-Knotens muss die value-Funktion auf einen zyklischen Graphen angewendet werden. Wie häufig ein Knoten ausgewertet wird, richtet sich dabei nach der sogenannten *Schleifentiefe* des Knotens.

Dabei gilt: Knoten mit Schleifentiefe 0 werden *höchstens* einmal ausgewertet und auf jede Auswertung eines Knotens mit Schleifentiefe d entfallen potentiell unendlich viele Auswertungen eines Knotens mit Schleifentiefe $d + 1$. Abbildung 4.1 demonstriert dies an einer verschachtelten Schleife. Knoten a in der inneren Schleife wird für jede Ausführung von Knoten b in der äußeren Schleife mehrfach ausgewertet.

Aus einer deklarativen Sichtweise heraus kann man wieder argumentieren, dass jeder Knoten mit Schleifentiefe $d + 1$ von d aus betrachtet eine unendliche Liste von Werten produziert, während umgekehrt von Schleifentiefe $d + 1$ aus betrachtet jeder Knoten aus d unendlich oft denselben Wert liefert.

Für das Beispiel in Abbildung 4.1 kann man so sagen, dass der Knoten b von Schleifentiefe 0 aus betrachtet die Liste $[4, 5, 7, 11, 19, ...]$ produziert. Wechselt man die Perspektive dagegen in die Schleifentiefe 1 hinein, so betrachtet man stets einzelne Werte von b, beispielsweise $b = 7$. Dagegen liefert a auf Schleifentiefe 2 nach wie vor eine unendliche Liste, die jedoch vom aktuellen Wert b abhängig ist. Für $b = 7$ wäre dies zum Beispiel: $[7, 8, 9, ...]$.

Analog dazu lassen sich die Werte von a auch aus der Schleifentiefe 0 heraus als Liste von Listen interpretieren: $[[4, 5, 6, ...], [5, 6, 7, ...], [7, 8, 9, ...], ...]$. Üblicherweise ist eine Betrachtung über mehr als eine Schleifentiefe hinweg aber nicht notwendig.

Die Betrachtung verläuft also stets relativ zu einem Bezugspunkt. Aus diesem Grunde wird die value-Funktion im Folgenden um die bisher fehlende Definition der Indexliste L ergänzt. Diese erfasst den Bezugspunkt dann auch formal.

4.3.2 Schleifensemantik

Definition 11 (Erweiterte value-Funktion). *Es sei δ die Schleifentiefe des Knoten v. Dann gilt:*

$$\text{value}_{[i_0,\ldots,i_d]}(v) = \begin{cases} [\text{value}_{[i_0,\ldots,i_d,i_\delta]}(v) \mid i_\delta \in \mathbb{N}_0], & \text{falls } d < \delta, \\ \text{value}_{[i_0,\ldots,i_\delta]}(v), & \text{falls } d > \delta. \end{cases}$$

Das heißt, je nach Indizes und Schleifentiefe bei der Auswertung, erhält man entweder einen spezifischen Wert oder eine Liste von Werten als Ergebnis von value_L. Für $d \ll \delta$ wird die Definition in mehreren Stufen angewendet, so dass man eine Liste von Listen erhält. Dieser Fall tritt in einem wohlgeformten Graphen jedoch nicht auf (siehe Abschnitt 4.4).

Definition 12 (θ-Knoten). *Ein Knoten $v = \theta(init, next)$ produziert eine Reihe von Werten und hat eine zugeordnete Schleifentiefe $v.depth = \delta$. Das produzierte Ergebnis ist definiert als:*

$$\begin{aligned} \text{value}_{[i_1,\ldots,i_{\delta-1},0]}(v) &= \text{value}_{[i_1,\ldots,i_{\delta-1}]}(v.init) \\ \text{value}_{[i_1,\ldots,i_{\delta-1},m]}(v) &= \text{value}_{[i_1,\ldots,i_{\delta-1},m-1]}(v.next) \end{aligned}$$

Anschaulich: der Wert mit Index 0 für die Schleifentiefe δ wird aus init initialisiert. Der Wert für den Index m definiert sich aus dem vorigen Ergebnis zum Index $m-1$. Ein Beispiel verdeutlicht dies:

$$\begin{aligned} v &\coloneqq \theta_1(2, 2+v) && \text{(Rekursive Definition.)} \\ \text{value}_{[]}(v) &= [2, 4, 6, \ldots] && \text{(Zugriff außerhalb der Schleife.)} \\ \text{value}_{[2]}(v) &= 6 && \text{(Zugriff innerhalb der Schleife.)} \\ \text{value}_{[2,1,0]}(v) &= 6 && \text{(Zugriff in verschachtelter Schleife.)} \end{aligned}$$

Der θ-Knoten setzt keine rekursive Definition des *next*-Wertes voraus. Damit sind Beispiele wie $\text{value}_{[]}(\theta(0,1)) = [0,1,1,1,\ldots]$ ebenso möglich.

Für die erweiterte value_L-Funktion ist es außerdem notwendig, dass *jeder* Knoten eine zugeordnete Schleifentiefe besitzt, da nur so erkannt werden kann, ob eine Kante die Schleife betritt oder verlässt. Beispielsweise lässt sich dem Knoten a aus Abbildung 4.1 die Schleifentiefe 2 zuordnen. Da bisher nur die Schleifentiefe der θ-Knoten bekannt ist, muss auch für die restlichen Knoten eine Schleifentiefe definiert werden. Es gilt:

Definition 13 (Schleifentiefe). *Für Knoten v mit $v.kind \neq \theta$ gilt:*

$$v.depth = \begin{cases} v.value.depth - 1, & \text{falls } v.kind = \eta, \\ \max\limits_{w \in v.deps} w.depth, & \text{falls } v.deps \neq \emptyset, \\ 0, & \text{sonst.} \end{cases}$$

Die Idee besteht darin, dass jeder Knoten v die Schleifentiefe $\text{depth}(v)$ an seine Benutzer vererbt, denn ein solcher Benutzer muss mindestens genauso häufig Werte produzieren.

Dies trifft auch auf Abhängigkeiten mit gemischten Schleifentiefen zu. Beispielsweise erhält Knoten a (Abbildung 4.1) die Schleifentiefe $2 = \max\{1, 2\}$, denn für jeden Wert des θ-Knotens lässt sich ein neues Ergebnis von a bestimmen, auch wenn sich die zweite Abhängigkeit b nur seltener ändert.

Die Ausnahme von dieser Regel bilden die η-Knoten, welche die Schleifentiefe um 1 herabsetzen. Sie selektieren genau einen Wert aus einer unendlichen Liste.

4.4 Korrektheit

Natürlich lassen sich Graphen konstruieren, deren Semantik undefiniert ist. Dies trifft insbesondere auf die Schleifendarstellung zu. Da sich die Betrachtung im Folgenden auf korrekte Graphen beschränkt, seien die wichtigsten Kriterien hierfür an dieser Stelle genannt.

Eingeschränkt zyklisch Jeder im Graph vorkommende Zyklus *muss* mindestens einen θ-Knoten v enthalten und durch dessen *next*-Abhängigkeit v.next verlaufen. Das heißt: entfernt man die *next*-Kanten der θ-Knoten, so erhält man einen azyklischen Graphen.

Konsistente Schleifentiefe Die als Attribut der θ-Knoten vorgegebenen Schleifentiefen müssen konsistent sein. Dies äußert sich darin, dass bei jedem η-Knoten v die Argumente v.cond und v.value *dieselbe Schleifentiefe* aufweisen müssen. Formal ausgedrückt muss $v.\text{cond.depth} = v.\text{value.depth}$ stets erfüllt sein.

Korrekter Schleifenzugriff Jeder Zugriff über eine Kante $(v, w) \in E$ mit $\Delta d = w.\text{depth} - v.\text{depth} > 0$ *muss* über einen η-Knoten erfolgen. Das heißt, es muss $v.\text{kind} = \eta$ gelten. Dies stellt sicher, dass stets eine Abbruchbedingung für die Auswertung angegeben ist. Darüber hinaus darf ein Zugriff per η-Knoten *höchstens* eine Schleifentiefe von $\Delta d = 1$ überbrücken.

5 Aufbau

5.1 Die Firm-Darstellung

Der Aufbau des vFirm-Programmes soll aus der Zwischensprache Firm heraus erfolgen. Diese wurde an der Universität Karlsruhe entwickelt [BBZ11; Lin02; TLB99] und stellt einen CFG in SSA-Form dar.

Auf den ersten Blick gibt es diverse Parallelen zwischen VSDG/PEG und Firm: Operationen werden als Knoten in einem Graphen dargestellt, welche durch Datenabhängigkeiten verbunden sind. Im Gegensatz zu den vorgestellten rein werteorientierten Zwischensprachen gibt es jedoch keine Gating-Funktionen. Ein Beispielgraph ist in Abbildung 5.1 zu sehen.

5.1.1 Konventionen

Wie bereits in Kapitel 1 angedeutet, werden in Firm Steuerflussgraph und Knoten für die Instruktionen in einem Graphen kombiniert. Das heißt, jeder Instruktions-Knoten ist in einem Grundblock eingebettet. Die Grundblöcke werden dabei selbst als Knoten mit v.kind = `block` dargestellt. Ansonsten gelten die in Abschnitt 2.3.1 vorgestellten Konventionen.

Endet ein Grundblock v in einem bedingten Sprung, dann wird die Bedingung des entsprechenden `cond`-Knotens auch direkt über das Attribut v.cond des Grundblockes eingeblendet (ansonsten ist der Wert $\perp$). Dies erleichtert die Formulierung von Algorithmen auf der Ebene des Steuerflussgraphen. `cond`

Wenn eine Instruktion mehr als ein Ergebnis zurückliefert, wird in der Firm-Darstellung für gewöhnlich ein Tupel als Rückgabewert verwendet und ein sogenannter `proj`-Knoten ist notwendig, um eines der Ergebnisse aus dem Tupel zu selektieren. Im Sinne einer einfacheren Darstellung wird dies hier durch die Kantenbeschriftung gekennzeichnet. `proj`

Beispielsweise „produziert“ der bedingte Sprung `cond` zwei Steuerflusswerte, die von den beiden Nachfolgeblöcken verwendet werden. Anstatt die Werte mittels `proj`-Knoten zu extrahieren, werden hier einfach die Kantenbeschriftungen e.kind $\in$ {*true, false*} verwendet (siehe Abbildung 5.1).

Start- und End-Block werden in den Abbildungen ebenfalls nicht explizit dargestellt, da Anfang und Ende der Funktion bereits an `start`- und `return`-Knoten ersichtlich sind. Die Zustandskanten werden gestrichelt dargestellt, Steuerflusskanten gepunktet.

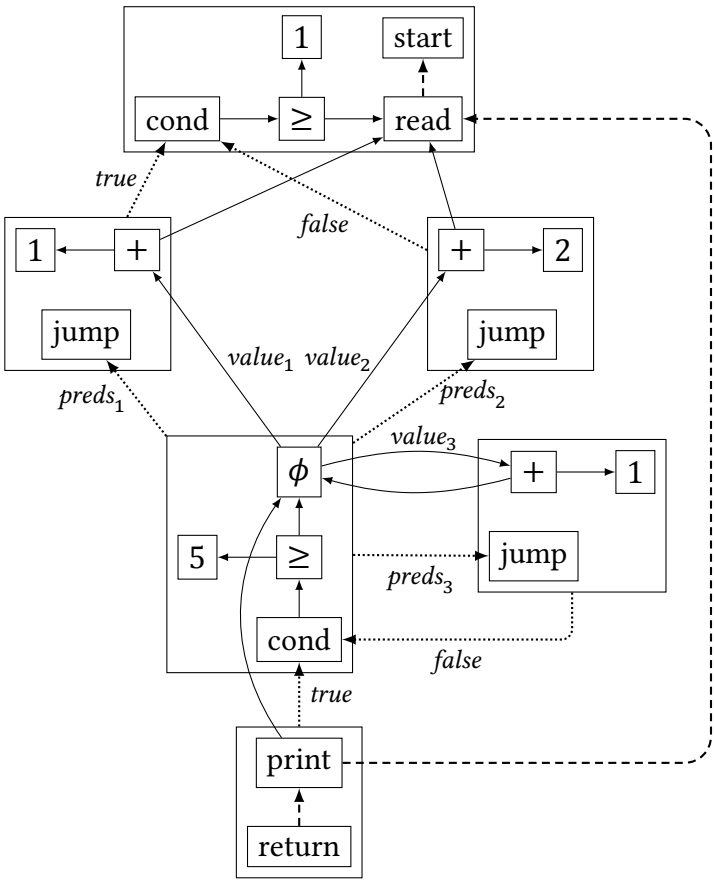


Abbildung 5.1: Beispielgraph in Firm

5.1.2 ϕ-Knoten

Kann ein Grundblock über mehrere Steuerflusskanten betreten werden, so ermöglichen ϕ-Knoten die Selektion eines Wertes abhängig vom Steuerfluss. Die ϕ-Knoten in einem Grundblock v mit $n = |v.\text{predEdges}|$ haben dabei genau n Parameter $value_1$, ..., $value_n$, so dass jeder Steuerflusskante ein Wert zugeordnet werden kann. Wird der Grundblock über $v.\text{predEdges}_i$ betreten, so selektiert der ϕ-Knoten den Wert $w.\text{values}_i$ ($i \in \{1, ..., n\}$).

Im Beispiel wird somit beim Rückwärtssprung über die mit $preds_3$ beschriftete Kante der addierte Schleifenzähler am Ende der mit $value_3$ beschrifteten Kante selektiert.

5.2 Grundüberlegungen

Ziel der Aufbau-Phase ist die Konstruktion eines äquivalenten vFirm-Programmes aus der Firm-Darstellung heraus. Da sich die Darstellung von vFirm und Firm sehr ähneln, sind einerseits keine Änderungen an der Infrastruktur notwendig und andererseits lässt sich der Aufbau durch Modifikation des existierenden Graphen erreichen.

Im Wesentlichen entfallen die Grundblöcke und damit der CFG, während gleichzeitig die Gating-Funktionen in Form neuer Knoten hinzukommen. Um mit der bestehenden Darstellung kompatibel zu bleiben, bleibt allerdings ein einfacher CFG bestehen. Dieser besteht lediglich aus Start- und End-Block, sowie einem einzigen weiteren Grundblock, welcher alle Knoten bis auf `start` und `end` enthält.

Damit besteht die wesentliche Aufgabe des Aufbaus darin, die ϕ-Knoten durch entsprechende γ- und θ-Knoten zu ersetzen, sowie η-Knoten für den Zugriff auf Schleifenwerte einzufügen. Alle weiteren Knoten und Abhängigkeiten entsprechen der bisherigen Firm-Darstellung, so dass im Anschluss lediglich noch der CFG aufgelöst werden muss.

5.3 Aufbaualgorithmus

Die Grundidee des Aufbaus ist aus dem technischen Bericht zum PEG-Aufbau [TSTL11] entnommen. Allerdings ist aufgrund der Firm-Ausgangsbasis ein veränderter Algorithmus notwendig, da wir keinen neuen Graphen aufbauen, sondern lediglich den Firm-Graphen modifizieren.

Davon abgesehen verarbeitet der Algorithmus beliebigen Steuerfluss, solange dieser reduzibel [HU74] ist. Der Steuerfluss muss also nicht vollkommen strukturiert sein, so dass eine Transformation auch trotz vorhergehender Optimierungen (oder expliziter Sprünge mittels `goto`) möglich ist.

5.3.1 Vorbereitung

Zur Vorbereitung der Transformation müssen einige Konstruktionen, welche in der neuen Darstellung (noch) nicht möglich sind, zu äquivalenten aber unterstützten Konstrukten vereinfacht werden. Bei Firm betrifft dies den `switch`-Knoten, welcher eine Werteselektion ähnlich der `switch`-Instruktion in C ermöglicht. Diese Knoten müssen zuerst durch eine entsprechende Kaskade von Verzweigungen ersetzt werden.

Wichtiger ist jedoch die Stelle, an der die Funktion verlassen wird. Da die Auswertungssemantik der neuen Darstellung ausgehend vom `return`-Knoten definiert wird, darf es nur einen eindeutigen `return`-Knoten geben. Dies lässt sich jedoch einfach bewerkstelligen, indem man einen zusätzlichen Grundblock mit `return`-Knoten erzeugt und die bestehenden `return`-Knoten durch Sprünge auf diesen Grundblock ersetzt. Eventuell muss der Rückgabewert mit einem ϕ-Knoten selektiert werden.

5.3.2 Aufbau der γ-Ausdrücke

Die ϕ-Knoten selektieren abhängig vom Steuerfluss einen spezifischen Wert aus einer Liste von Werten. Der CFG und damit das Konzept eines Steuerflusses ist in der neuen Darstellung jedoch nicht mehr vorhanden.

Dennoch lässt sich der gewählte Ablaufpfad noch rekonstruieren, indem man die Bedingungen der zuvor verwendeten bedingten Sprünge (mittels `cond`-Knoten) betrachtet. Eine Selektion des korrespondierenden Wertes ist dann mit Hilfe dieser Bedingungen und einer Kaskade von γ-Knoten möglich.

5.3.3 Grundidee

Der Aufbau der γ-Knoten für einen Grundblock v lässt sich in zwei Aufgaben unterteilen:

1. Erfasse alle Bedingungen, die einen Einfluss auf die Steuerflusskante haben, durch welche v betreten wird.
2. Baue einen Selektionsgraphen aus γ-Knoten auf, der abhängig von den Wahrheitswerten der `cond`-Bedingungen einen Wert des alten ϕ-Knotens selektiert.

Beide Aufgaben lassen sich gemeinsam beim Traversieren des Steuerflussgraphen erledigen. Wir beginnen dabei an den Vorgängerblöcken von v und bewegen uns im Steuerflussgraphen entgegen der Ausführungsrichtung. Bei jeder Verzweigung werden die zuvor berechneten Selektionsgraphen der Zielblöcke mit einem γ-Knoten kombiniert. Es handelt sich also um einen Teile-und-Herrsche-Algorithmus.

Der Arbeitsbereich

Man kann den Teil des Graphen, der betrachtet werden muss, in beide Richtungen begrenzen. Die triviale Bedingung ist dabei die Erreichbarkeit von v: Ein Grundblock, der v nicht erreicht, muss auch nicht betrachtet werden.

Für die andere Richtung betrachten wir den direkten Dominator $v_0 = \text{idom}(v)$ von v. Die Dominanzrelation bezieht sich hierbei auf den Steuerflussgraphen mit Kanten in Ausführungsrichtung (siehe 2.3.3) und damit ist die Ausführung von v_0 eine notwendige Bedingung für die Ausführung von v.

Grundblöcke, die von v_0 aus nicht erreicht werden können, dürfen also ignoriert werden. Sie können v entweder gar nicht oder nur durch v_0 erreichen und sind damit für die Frage, über welche Steuerflusskante v betreten wird, irrelevant.

Listing 1 Initialisierung der γ-Konstruktion

```
1:  function BuildGammas(S, ϑ : S → V, v_0)
2:      C ← {e.src.block | e ∈ S} ∪ {v_0}
3:      while ∃p ∈ (⋃_{v∈C\{v_0}} v.preds) \ C do
4:          C ← C ∪ {p}
5:      end while
6:
7:      for all e ∈ S do
8:          e.incoming ← ϑ(e)
9:      end for
10:
11:     return WalkCFG(v_0, C)
12: end function
```

Initialisierung

Listing 1 stellt den notwendigen Algorithmus dar. Der BuildGammas-Funktion werden dafür die relevanten Steuerflusskanten S, eine Werte-Zuordnung $\vartheta : S \to V$, sowie der direkte Dominator v_0 übergeben. Dies erlaubt die spätere Verwendung der Funktion in anderen Zusammenhängen.

C

Der Algorithmus traversiert den Graphen ausgehend von v_0, weshalb die Erreichbarkeit von v_0 aus trivialerweise erfüllt ist. Die zweite Einschränkung an den Arbeitsbereich, nämlich dass eine der Kanten S erreichbar ist, wird durch die Fixpunktiteration ab Zeile 3 vorbereitet. Hierbei werden die Grundblöcke C gesammelt, welche eine der Kanten $e \in S$ erreichen können. Dies erlaubt später den Abbruch bei Grundblöcken w mit $w \notin C$.

incoming
outgoing

Wir beginnen dann mit WalkCFG die Traversierung des Graphen, wobei wir post-order die Selektionsgraphen zusammensetzen. Um die Teilergebnisse zu transportieren, werden die incoming- und outgoing-Attribute verwendet. Ein Grundblock w mit Verzweigung erhält so über e.incoming die unfertigen Selektionsgraphen von den Nachfolgerblöcken. Sie sind den Kanten zu diesen Blöcken zugeordnet. Nachdem beide Selektionsgraphen durch die Konstruktion eines γ-Knotens kombiniert wurden, wird das Ergebnis in w.outgoing gespeichert und gegebenenfalls über die Steuerflusskanten an die Vorgänger-Blöcke weitergegeben.

Bei der Initialisierung werden die unmittelbar bekannten Werte $\vartheta(e)$ für die Kanten $e \in S$ über das incoming-Attribut dieser Kanten gespeichert (Zeile 7). Für die anderen Attribute wird der Standardwert $\bot$ angenommen.

Im Anschluss wird die in Listing 2 dargestellte WalkCFG-Funktion aufgerufen, welche die eigentliche Konstruktion durchführt und nun genauer betrachtet wird.

Listing 2 Konstruktion der γ-Knoten

```
1:  function WalkCFG(v, C)
2:      if (v ∉ C) or v.isMarked then
3:          return ⊥
4:      end if
5:
6:      if v.outgoing = ⊥ then
7:          v.isMarked ← true
8:          for all e ∈ v.succEdges : e.incoming = ⊥ do
9:              e.incoming ← WalkCFG(e.dst, C)
10:         end for
11:         v.isMarked ← false
12:
13:         I ← {e ∈ v.succEdges | e.incoming ≠ ⊥}
14:         if I = {e} then
15:             v.outgoing ← e.incoming
16:         else
17:             assert |I| = 2
18:             t ← {e.incoming | e ∈ I, e.kind = true}
19:             f ← {e.incoming | e ∈ I, e.kind = false}
20:             v.outgoing ← new γ(v.cond, t, f)
21:         end if
22:     end if
23:
24:     return v.outgoing
25: end function
```

Traversierung

Abbruch

Es gibt dabei zwei Fälle, in denen für den Grundblock v kein Selektiongraph bestimmt werden kann. Schon genannt wurde die Einschränkung des Arbeitsbereiches. Bei $v \notin C$ ist keine Kante S erreichbar. Der zweite Fall betrifft Zyklen im Steuerflussgraphen. Da wir Schleifen später getrennt verarbeiten müssen, ignoriert der Algorithmus diese zunächst und beschränkt sich auf den azyklischen Schleifenkörper. In beiden Fällen wird $\bot$ zurückgeliefert (Zeile 2).

Rekursion

Ansonsten erfolgt zunächst die Rekursion (Zeile 9), falls der Selektionsgraph v.outgoing für den Grundblock noch unbekannt ist. Dabei werden ebenfalls nur solche Folgeblöcke verarbeitet, für die noch kein Wert bekannt ist. Das isMarked-Attribut dient hier zur Markierung aller Grundblöcke im gerade betrachteten Pfad. Dadurch kann in Zeile 2 erkannt werden, ob ein Zyklus vorliegt.

Verarbeitung

Im Anschluss an die Rekursion müssen die Ergebnisse e.incoming an den Kanten e zu den Folgeblöcken vorliegen. Handelt es sich dabei um nur einen Wert (weil nur eine Kante exis-

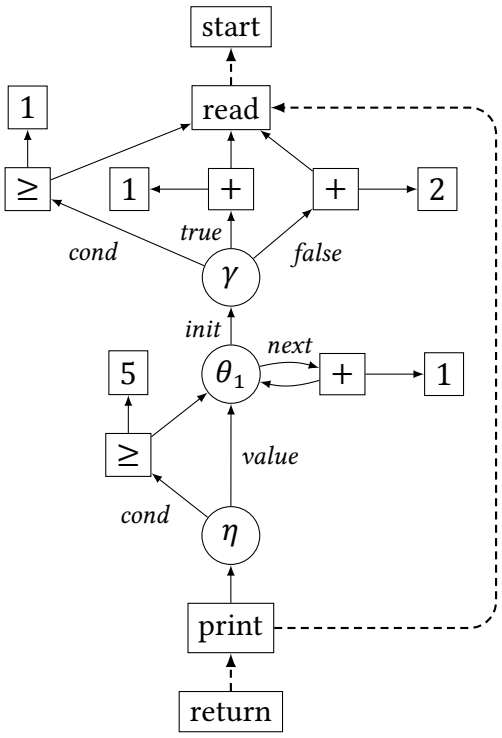


Abbildung 5.2: Äquivalenter vFirm-Graph zu Abbildung 5.1

tiert, der Arbeitsbereich verlassen wird oder ein Zyklus vorliegt), dann wird dieser direkt als Ergebniswert v.outgoing übernommen (Zeile 15).

Ansonsten muss in diesem Grundblock eine Verzweigung stattfinden, wobei für beide Wahrheitswerte ein Selektionsgraph bekannt ist. Diese werden den jeweiligen Kanten entnommen und dann zur Konstruktion eines γ-Knotens verwendet, welcher die Bedingung der in v vorliegenden Verzweigung verwendet (Zeile 20). Der γ-Knoten bildet dann die Wurzel des Selektionsgraphen für v.

Wenn der erste Aufruf von WalkCFG auf v_0 zurückkehrt, dann ist die Verarbeitung abgeschlossen. Der vollständige Selektionsgraph wird als Ergebnis zurückgeliefert. Abschluss

5.3.4 Aufbau der θ-Knoten

Ebenso wie die γ-Kaskaden, ersetzen θ-Knoten die ϕ-Knoten des „alten“ CFG. Sie sind immer dann erforderlich, wenn sich der ϕ-Knoten in einem Schleifenkopf befindet, das heißt in einem Grundblock mit Vorgängern inner- und außerhalb der Schleife.

Der Wert, der vom ϕ-Knoten beim Betreten der Schleife selektiert wird, dient als Initialisierungswert des θ-Knotens. Entsprechend dient der Wert, den der ϕ-Knoten beim Rücksprung in den Schleifenkopf selektiert, als Wiederholungswert des θ-Knotens. Wenn in einem der

beiden Fälle mehrere Kanten in Betracht kommen, sind weiterhin γ-Knoten zur Selektion notwendig.

Listing 3 Ersetzen von ϕ-Knoten

```
1:  procedure ReplacePhi(v)
2:      assert v.kind = φ
3:
4:      v_b ← v.block
5:      ϑ  ← {(v_b.predEdges_1, v.value_1), ..., (v_b.predEdges_n, v.value_n)}
6:
7:      I  ← {e ∈ v_b.predEdges | e.dst.block ∈ v_b.loop.blocks}
8:      O  ← v_b.predEdges \ I
9:
10:     v_r ← BuildGammas(I, ϑ, dom({e.dst | e ∈ I}))
11:     if O ≠ ∅ then
12:         v_o ← BuildGammas(O, ϑ, dom({e.dst | e ∈ O}))
13:         d  ← v_b.loop.depth
14:         v_r ← new θ_d(v_o, v_r)
15:     end if
16:
17:     Replace(v, v_r)
18: end procedure
```

Die in Listing 3 dargestellte ReplacePhi-Prozedur berücksichtigt diese Umstände und ersetzt beliebige ϕ-Knoten v entweder durch einen γ- oder θ-Knoten, abhängig davon, ob sich der Knoten in einem Schleifenkopf befindet.

Für die Klassifizierung der Steuerflusskanten v_b.predEdges, durch welche v_b betreten werden kann, wird der in Abschnitt 2.3.4 vorgestellte Schleifenbaum verwendet. v_b.loop.blocks umfasst alle Grundblöcke der v_b umschließenden Schleife und damit lässt sich zuordnen, entlang welcher Kanten I ein Rücksprung innerhalb der Schleife stattfindet und durch welche Kanten O die Schleife betreten wird (Zeile 7+8).

In jedem Fall wird mit Hilfe der BuildGammas-Funktion ein γ-Graph zur Selektion eines Wertes aus den inneren Kanten aufgebaut, der entweder direkt als Ersatz für den ϕ-Knoten dient oder durch einen θ-Knoten ergänzt wird. Als Start-Knoten für BuildGammas dient dabei der nächste gemeinsame Dominator der jeweiligen Ursprungsblöcke (siehe 2.3.3).

Die Ergänzung um einen θ-Knoten erfolgt dann, wenn tatsächlich äußere Kanten gefunden wurden ($O \neq \emptyset$). In diesem Fall handelt es sich bei v_b um den Schleifenkopf. Für die äußeren Kanten wird dann ein weiterer γ-Graph konstruiert und im Anschluss der θ-Knoten, welcher den ϕ-Knoten ersetzen soll (Zeile 14).

Das Beispiel in den Abbildungen 5.1 und 5.2 demonstriert dieses Vorgehen. Der Schleifenkopf mit dem ϕ-Knoten besitzt zwei Vorgänger außerhalb der Schleife ($preds_1$ und $preds_2$),

sowie einen innerhalb ($preds_3$). Dementsprechend wird mit BuildGammas ein γ-Ausdruck zur Selektion des Initialisierungswertes anhand der Verzweigung im ersten Grundblock konstruiert. Der next-Wert kann dagegen unmittelbar vom ϕ-Knoten übernommen werden.

5.3.5 ϕ-Knoten gemeinsam verarbeiten

Mit der ReplacePhi-Prozedur lassen sich alle ϕ-Knoten durch einen Graphen aus γ-Knoten und/oder θ-Knoten ersetzen. Für ϕ-Knoten, die sich in demselben Grundblock befinden, ist die Struktur des konstruierten Graphen dabei stets identisch, denn sie richtet sich nach dem umliegenden CFG. Die resultierenden Graphen unterscheiden sich lediglich in den selektierten Werten. Es bietet sich daher an, die Konstruktion des Selektionsgraphen nur einmal vorzunehmen und diesen anschließend für jeden ϕ-Knoten zu replizieren.

Eine naheliegende Lösung ist es, die ϕ-Knoten eines Grundblockes zunächst zusammenzufassen, so dass durch den ϕ-Knoten keine individuellen Werte selektiert werden sondern Tupel, die aus mehreren Werten bestehen. Abbildung 5.3 stellt dies an einem Beispiel dar. Beide ϕ-Knoten befinden sich hierbei in demselben Grundblock. Die hier als MakeTuples beschriftete Prozedur fasst dabei Werte zusammen, welche aus denselben Ursprungsblöcken stammen, zum Beispiel v_1 und v_4.

Damit verbleibt in jedem Grundblock höchstens ein ϕ-Knoten, so dass die ReplacePhi-Prozedur nur für diesen ϕ-Knoten einen Selektionsgraphen aufbauen muss. Anschließend kann der Selektionsgraph durch die CloneTuples-Prozedur wieder zerlegt werden.

5.3.6 Ergänzung der η-Knoten

Wenn alle ϕ-Knoten im Graphen ersetzt wurden, sind immer noch keine η-Knoten vorhanden. Das heißt, die Abbruchbedingungen der Schleifen werden bis zu diesem Zeitpunkt noch nicht berücksichtigt. Sie sind aber immer dann erforderlich, wenn von außen auf eine Schleife zugegriffen wird.

Solche Zugriffe stellen sich als Datenabhängigkeiten zwischen Blöcken dar, wobei sich der Zielblock in einer tieferen Schleife befindet als der Quellblock.[1] Diese Kanten zu finden, ist mit Hilfe des Schleifenbaumes relativ einfach. Die Schwierigkeit besteht daher im Bestimmen der Abbruchbedingung.

Bedingung

Glücklicherweise weist das Vorgehen erneut Ähnlichkeiten mit dem Aufbau der γ-Graphen auf. Es muss abhängig davon, welche CFG-Kante erreicht wird, ein Wert selektiert werden: nämlich true für jede Kante, welche die Schleife verlässt und false für Kanten, welche erneut den Schleifenkopf betreten, um eine weitere Iteration zu beginnen. Wieder lässt sich BuildGammas zur Konstruktion verwenden. Das Vorgehen wird in der BuildBreakCond-Funktion in Listing 4 dargestellt.

[1] Gemäß Abschnitt 2.3.4 wird außerhalb von Schleifen eine Tiefe von 0 angenommen.

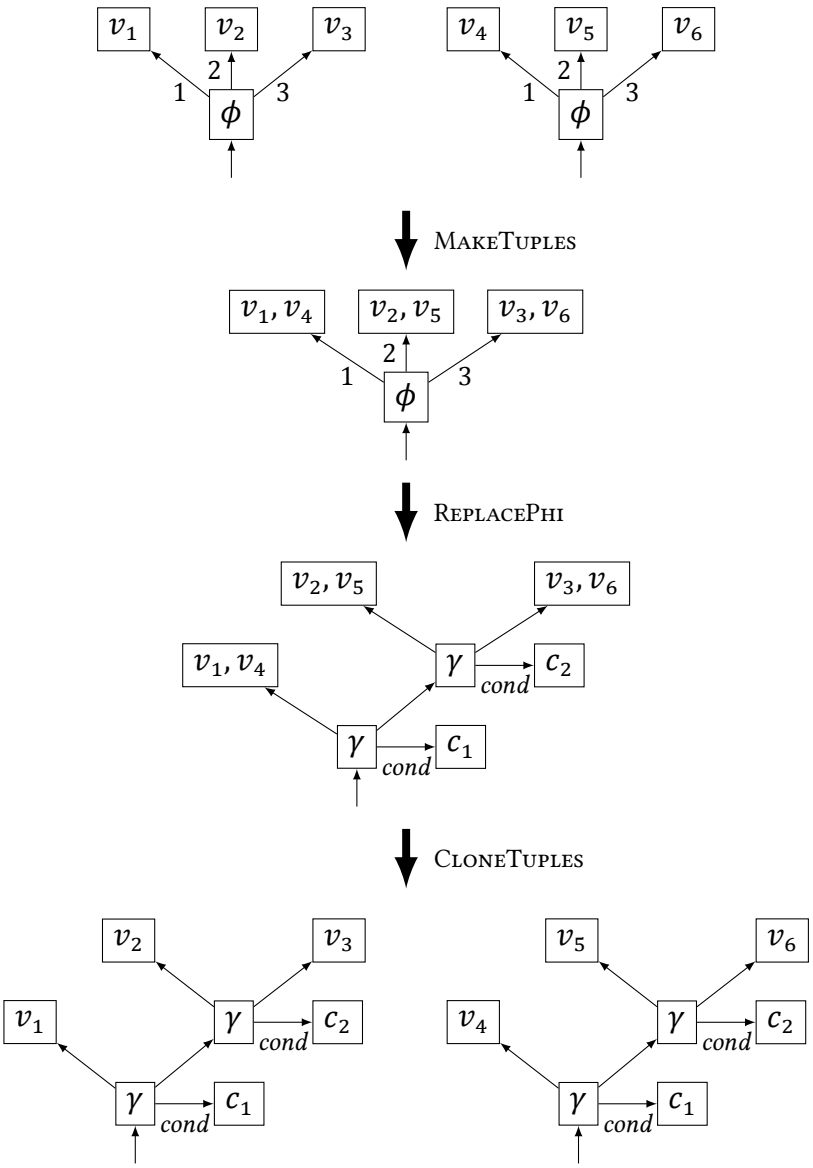


Abbildung 5.3: Gemeinsame Verarbeitung mit Tupeln

Listing 4 Aufbau der η-Knoten

```
1:  procedure InsertEtaNodes(G = (V, E, v_s, v_e))
2:      for all e = (v, w) ∈ E : v.block ∉ w.block.loop.blocks do
3:          if L.breakCond = ⊥ then
4:              L.breakCond ← BuildBreakCond(L)
5:          end if
6:          e.dst ← new η(L.breakCond, w)
7:      end for
8:  end procedure
9:
10: function BuildBreakCond(L)
11:     X ← {e ∈ ⋃_{v∈L.blocks} v.succEdges | e.src ∉ L.blocks}
12:     Y ← {e ∈ L.header.predEdges | e.dst ∈ L.blocks}
13:     ϑ ← λb.(b ∈ X)
14:     return BuildGammas(X ∪ Y, ϑ, L.header)
15: end function
```

Dabei werden zunächst die genannten Kanten bestimmt. Kanten, welche die Schleife L verlassen, bilden die Menge X. Die Rückwärtskanten zum Schleifenkopf bilden Y. Für jede dieser Kanten lässt sich ein boolescher Wert bestimmen, welcher durch den Lambda-Ausdruck für ϑ berechnet wird: nur die Kanten aus X erhalten dabei den Wert true. Mit diesen Parametern konstruiert BuildGammas genau die Abbruchbedingung der Schleife.

Um auf L.header zugreifen zu können, muss ein eindeutiger Schleifenkopf existieren. Wie in Abschnitt 2.3.4 erwähnt, setzt dies einen reduziblen CFG [HU74] voraus. Diese Einschränkung ist jedoch zu vertreten, da sich ein irreduzibler CFG stets in einen reduziblen CFG umwandeln lässt, indem man einige Grundblöcke der Schleife dupliziert und zusammen mit den zusätzlichen Einsprungspunkten aus dieser hinauszieht.

Konstruktion

Nachdem die Schleifenbedingung bekannt ist, ist der Aufbau der erforderlichen η-Knoten relativ einfach. Dies ist in der InsertEtaNodes-Prozedur dargestellt. Zunächst werden genau jene Abhängigkeitskanten gesucht, die von außen auf eine Schleife zugreifen (Zeile 2). Für die Schleife wird dann jeweils – soweit noch nicht geschehen – die Abbruchbedingung bestimmt und zwischengespeichert. Schließlich wird der Zielknoten der Kante durch einen η-Knoten ersetzt, welcher die Abbruchbedingung mit einbezieht.

5.3.7 Abschluss

Mit den vorgestellten Funktionen und Prozeduren steht das nötige Handwerkszeug für die Transformation nach vFirm bereit. Der abschließende Algorithmus, welcher den gesamten Ablauf kapselt, ist in Listing 5 dargestellt.

Listing 5 Transformation in die neue Darstellung

```
1:  procedure FirmToVFirm(G = (V, E, v_s, v_e))
2:      // MakeTuples(G)
3:      for all v ∈ V : v.kind = φ do
4:          ReplacePhi(v)
5:      end for
6:      // CloneTuples(G)
7:      InsertEtaNodes(G)
8:
9:      v_b ← new Block()
10:     v_e.preds ← {v_b}
11:     v_b.preds ← {v_s}
12:
13:     for all v ∈ V : v.block ∉ {v_s, v_e} do
14:         v.block ← v_b
15:     end for
16: end procedure
```

Die Zeilen 2–7 wenden die bereits dargestellten Algorithmen auf den Graphen an. Die in Abschnitt 5.3.5 vorgestellte Verwendung der Tupel ist dabei nicht zwingend erforderlich.

Nach der eigentlichen Verarbeitung wird noch der CFG entfernt. Dabei bleiben Start-Block v_s und End-Block v_e bestehen. Knoten, die nicht einem dieser beiden Blöcke angehören, werden in einen neuen Grundblock v_b verschoben, welcher den eigentlichen Graphen repräsentiert (Zeile 14). Die Vorgängerblöcke werden entsprechend modifiziert, so dass nur noch ein linearer Steuerflusspfad durch den Graphen führt.

6 Abbau

Ziel des Abbaus ist die Konstruktion eines äquivalenten Firm-Graphen aus der (optimierten) vFirm-Darstellung. Der beim Aufbau aufgelöste Steuerflussgraph muss hierfür neu aufgebaut werden.

Im Gegensatz zur Firm-Darstellung, die in Bezug auf den Steuerfluss stets nur die Grundblöcke betrachtet, muss die Ausführung bei vFirm für jeden Knoten individuell betrachtet werden. Ein Knoten wird genau dann ausgewertet, wenn sein Wert benötigt wird. Es müssen also die Pfade durch den Graphen betrachtet werden, die zu diesem Knoten führen. Genauer gesagt, betrifft dies die Gating-Knoten auf diesen Pfaden, da nur diese eine selektive Abfrage ihrer Abhängigkeiten erlauben.

Es liegt nahe, diese Auswertungsbedingung näher zu formalisieren und einen direkten Zusammenhang zu den booleschen Werten im Graphen herzustellen. Haben dann zwei Knoten dieselbe Auswertungsbedingung, so ist dies ein starkes Indiz dafür, dass beide gemeinsam ausgewertet werden und daher im CFG demselben Grundblock zugeordnet werden können. Es ist jedoch auch zu beachten, dass Abhängigkeiten zwischen diesen Knoten eine Zuordnung zu demselben Grundblock verhindern können. Für eine erste Klassifizierung der Knoten sind solche Auswertungsbedingungen aber dennoch ein sehr nützliches Werkzeug und daher ein Grundbaustein des Abbau-Algorithmus.

6.1 Gating-Pfade und -Bedingungen

Auch beim Abbau spielt die Dominanzrelation eine wichtige Rolle (vgl. Abschnitt 2.3.3). Wir betrachten hierbei den `return`-Knoten als Start-Knoten und verwenden die normalen Abhängigkeitskanten zum Aufbau des Dominanzbaumes. Die Baumstruktur des Dominanzgraphen ist dabei immer gegeben, da jeder Knoten im vFirm-Graphen vom `return`-Knoten aus erreichbar sein muss.

Ob und wann ein Knoten ausgewertet wird, lässt sich durch *Gating-Pfade* und *-Bedingungen* erfassen. Sie können aufgrund ihrer Definition systematisch anhand des Dominanzbaumes aufgebaut werden und dennoch alle Pfade im Graphen abdecken.

Definition 14 (Gating-Pfad). *Ein Pfad* $P = (v_1, v_2) \to (v_2, v_3) \to \ldots \to (v_{n-1}, v_n)$ *mit* $v_i \in V, i \in \{1, \ldots, n\}$ *durch den Graphen* $G = (V, E)$ *heißt Gating-Pfad, wenn* $\{v_1, \ldots, v_{n-1}\} \subseteq D(v_1)$.

Das Verfahren, um die Gating-Pfade mit ihren entsprechenden Bedingungen zu verknüpfen, stammt aus der Dissertation von Lawrence [Law07]. Aufgrund der dort verwendeten rekursiven Schleifendarstellung werden dort keine θ- oder η-Knoten betrachtet und wir beschränken uns daher auch zunächst auf azyklische Graphen.

Im Rahmen dieser Arbeit werden Gating-Bedingungen als aussagenlogische Formeln aufgefasst und üblicherweise in disjunktiver Normalform dargestellt. Die Elementaraussagen dieser Formeln stützen sich dabei auf die im Graphen vorkommenden Bedingungs-Knoten.

Definition 15 (Bedingungs-Knoten). *Ein Knoten v heißt* Bedingungs-Knoten *genau dann, wenn es einen γ-Knoten w gibt, mit $w = \gamma(v, value_{\text{true}}, value_{\text{false}})$. Dabei ist notwendigerweise $v.type = \mathbb{B}$.*

Kurz gesagt: jeder Knoten der als Bedingung eines γ-Knotens auftritt gilt als Bedingungs-Knoten. Für die Gating-Bedingungen ergibt sich dann:

Definition 16 (Gating-Bedingung). *Eine Gating-Bedingung ist eine aussagenlogische Formel, die durch Verknüpfung mehrerer Elementaraussagen entsteht. Die Elementaraussage a ist genau dann wahr, wenn der korrespondierende Bedingungs-Knoten a bei der Auswertung des Graphen den Wert* true *annimmt, also* $\text{value}_L(a) = \text{true}$ *gilt.*

Es ist bereits zu erkennen, dass die Elementaraussage für $L \neq \emptyset$ – also innerhalb einer Schleife – abhängig von der aktuellen Iteration verschiedene Werte annehmen kann, für $L = \emptyset$ jedoch statisch ist. Solange keine Schleifen betrachtet werden, besitzt also jede durch einen Knoten berechnete Bedingung eine Elementaraussage mit eindeutigem Wert.

6.1.1 Verknüpfung von Gating-Pfad und -Bedingung

Gating-Bedingungen beschreiben, wann ein bestimmter Gating-Pfad oder eine Menge mehrerer Gating-Pfade ausgewertet wird. Sie lassen sich ausgehend von den Kanten definieren:

Definition 17 (Gating-Bedingungen für Kanten). *Für eine Kante $e = (v, w)$ lässt sich eine Gating-Bedingung* $\text{gc}(e)$ *definieren als:*

$$\text{gc}(e) = \text{gc}(v, w) = \begin{cases} c, & \text{falls } v.kind = \gamma \text{ und } w = v.\text{true mit } c = v.cond, \\ \bar{c}, & \text{falls } v.kind = \gamma \text{ und } w = v.\text{false mit } c = v.cond, \\ \text{true}, & \text{sonst.} \end{cases}$$

Die Bedingung $\text{gc}(e)$ verknüpft also die true- und false-Kanten eines γ-Knotens mit einer entsprechenden Bedingung. Alle anderen Kanten werden ohne Bedingung ausgewertet, so dass hier $\text{gc}(e) = \text{true}$ gilt.

Damit ein ganzer Gating-Pfad ausgewertet wird, muss jede Bedingung entlang des Pfades erfüllt sein. Die Bedingung ergibt sich also durch Konjunktion der Einzelbedingungen.

Definition 18 (Gating-Bedingungen für Pfade). *Die Gating-Bedingung* $\mathrm{gc}(P)$ *eines einzelnen Gating-Pfades* $P = (v_1, v_2) \to \ldots \to (v_{n-1}, v_n)$ *ergibt sich aus den Bedingungen der Einzelkanten folgendermaßen:*

$$\mathrm{gc}(P) = \bigwedge_{e \in P} \mathrm{gc}(e)$$

Knoten können jedoch auch über mehrere Gating-Pfade miteinander verbunden sein. Dabei muss die Auswertung des Knotens genau dann erfolgen, wenn mindestens einer der Gating-Pfade zu diesem Knoten ausgewertet wird. Entsprechend ergibt sich:

Definition 19 (Gating-Bedingungen für Pfadmengen). *Die Gating-Bedingung* $\mathrm{gc}(M)$ *für eine Pfadmenge* $M = \{P_1, \ldots, P_n\}$ *ergibt sich aus den Gating-Bedingungen der Pfade wie folgt:*

$$\mathrm{gc}(M) = \bigvee_{P \in M} \mathrm{gc}(P)$$

Im Endeffekt möchten wir wissen, wann ein Knoten w ausgewertet wird, unter der Voraussetzung, dass die Auswertung seines direkten Dominators $v = \mathrm{idom}(w)$ erfolgt. Dies ermöglicht es später, die Knoten ausgehend von der Wurzel des Dominanzbaumes und entsprechend ihrer relativen Bedingungen zu klassifizieren.

Es ist klar, dass die entsprechende Bedingung bestimmt werden kann, wenn alle Pfade zwischen v und w betrachtet und deren Bedingungen verknüpft werden. Aufgrund der Dominanzbeziehung zwischen v und w können dies nur Gating-Pfade sein, so dass wir auf die bestehenden Definitionen zurückgreifen können.

Definition 20 (Gating-Bedingung zwischen Knoten und direkten Dominator). *Für einen Knoten* w *mit direkten Dominator* $v = \mathrm{idom}(w)$ *gilt:*

$$\mathrm{gc}(v \Rightarrow w) = \mathrm{gc}(\{P \mid P \textit{ ist ein (Gating)-Pfad von } v \textit{ nach } w\})$$

6.1.2 Gating-Bedingungen am Beispiel

Abbildung 6.1 soll als Beispiel für die Bestimmung der Gating-Bedingungen dienen. Normale Linien stellen Datenkanten dar, während die gepunkteten Linien den Dominanzbaum darstellen.

Für die Berechnung von $\mathrm{gc}(\gamma_1 \Rightarrow u)$ müssen die Gating-Pfade $P_1 = (\gamma_1, u)$ und $P_2 = (\gamma_1, \gamma_2) \to (\gamma_2, u)$ betrachtet werden. Es gilt:

$$\begin{aligned} \mathrm{gc}(\gamma_1 \Rightarrow u) &= \mathrm{gc}(P_1) \vee \mathrm{gc}(P_2) \\ &= \mathrm{gc}(\gamma_1, u) \vee (\mathrm{gc}(\gamma_1, \gamma_2) \wedge \mathrm{gc}(\gamma_2, u)) \\ &= A \vee \bar{A}B = A \vee B \end{aligned}$$

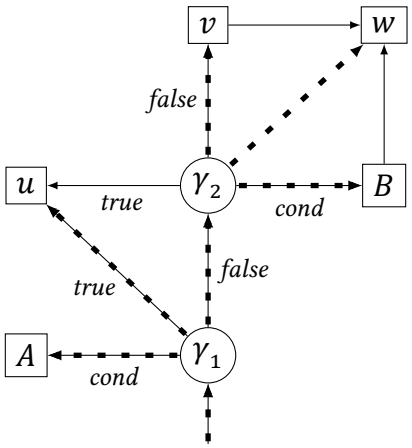


Abbildung 6.1: Beispielgraph für Gating-Bedingungen

Entsprechend erhält man für $\mathrm{gc}(\gamma_2 \Rightarrow w)$ mit den Gating-Pfaden $P_3 = (\gamma_2, v) \to (v, w)$ und $P_4 = (\gamma_2, B) \to (B, w)$:

$$\begin{aligned}\mathrm{gc}(\gamma_2 \Rightarrow w) &= \mathrm{gc}(P_3) \vee \mathrm{gc}(P_4) \\ &= (\mathrm{gc}(\gamma_2, v) \wedge \mathrm{gc}(v, w)) \vee (\mathrm{gc}(\gamma_2, B) \wedge \mathrm{gc}(B, w)) \\ &= (\bar{B} \wedge \mathrm{true}) \vee (\mathrm{true} \wedge \mathrm{true}) \\ &= \mathrm{true}\end{aligned}$$

Der Vollständigkeit halber sind alle Bedingungen $\mathrm{gc}(\mathrm{idom}(x) \Rightarrow x)$ hier aufgezählt:

$$\begin{aligned}\mathrm{gc}(\gamma_1 \Rightarrow A) &= \mathrm{true} & \mathrm{gc}(\gamma_2 \Rightarrow v) &= \bar{B} \\ \mathrm{gc}(\gamma_1 \Rightarrow u) &= A \vee \bar{A}B & \mathrm{gc}(\gamma_2 \Rightarrow w) &= \mathrm{true} \\ \mathrm{gc}(\gamma_1 \Rightarrow \gamma_2) &= \bar{A} & \mathrm{gc}(\gamma_2 \Rightarrow B) &= \mathrm{true}\end{aligned}$$

Hieraus lassen sich Bedingungen zwischen beliebigen Knoten herleiten, sofern einer den anderen dominiert. Beispielsweise:

$$\mathrm{gc}(\gamma_1 \Rightarrow v) = \mathrm{gc}(\gamma_1 \Rightarrow \gamma_2) \wedge \mathrm{gc}(\gamma_2 \Rightarrow v) = \bar{A}\bar{B}$$

Für die Konstruktion sind jedoch nur die Bedingungen zwischen Knoten und direktem Dominator von Relevanz.

6.1.3 Gating-Bedingungen vereinfachen

Im Folgenden wird angenommen, dass die aussagenlogischen Formeln während der Konstruktion direkt vereinfacht werden. Hierdurch lassen sich gegebenenfalls unnötige Verzweigungen im resultierenden Programm einsparen.

Ziel ist die Bestimmung einer disjunktiven Minimalform. Leider handelt es sich hierbei um ein NP-vollständiges Problem. Für diese Arbeit wurde daher ein einfaches heuristisches Verfahren verwendet. Die Bedingungen liegen nach der Konstruktion in einer disjunktiven Form vor und werden dann wiederholt mit Hilfe der folgenden zwei Regeln vereinfacht:

1. $aF \vee \ldots \vee \bar{a}F \vee \ldots \vee G = (a \wedge \bar{a}) \vee F \vee \ldots \vee G = F \vee \ldots \vee G$
2. $F \vee \ldots \vee FG \vee \ldots \vee H = F \vee \ldots \vee H$

Dabei stellen F, G und H beliebige aussagenlogische Formeln dar.

6.1.4 Gating-Bedingungen berechnen

Der Aufbau der Gating-Bedingungen kann – wie bereits erwähnt – systematisch von den Blättern des Dominators bis zur Wurzel erfolgen. Das Ziel ist dabei die Bestimmung der Gating-Bedingungen $\mathrm{gc}(\mathrm{idom}(x), x)$.

Dazu werden alle *minimalen* Gating-Pfade $(v_1, v_2) \rightarrow (v_2, v_3) \rightarrow \ldots \rightarrow (v_{n-1}, v_n)$ bestimmt. Dies sind Gating-Pfade, bei denen der Teilpfad $(v_2, v_3) \rightarrow \ldots \rightarrow (v_{n-1}, v_n)$ ohne die erste Kante selbst kein Gating-Pfad mehr ist. Die Vollständigkeit des Verfahrens lässt sich dabei durch strukturelle Induktion zeigen.

Induktionsanfang

Für ein Blatt v im Dominanzbaum lassen sich die hier beginnenden minimalen Gating-Pfade leicht aufzählen. Jede von v ausgehende Kante ist tatsächlich auch ein Gating-Pfad, denn per Definition muss der letzte Knoten eines Gating-Pfades *nicht* dominiert werden (selbst wenn dieser nur eine Kante umfasst).

Längere Gating-Pfade können nicht von v ausgehen, denn hierfür wäre ein von v dominierter Knoten erforderlich, der nicht existieren kann. Es verbleiben also lediglich die ausgehenden Kanten von v, deren Bedingungen direkt angegeben werden können (siehe Definition 17).

Induktionsvoraussetzung

Um die Vollständigkeit der Berechnung für einen Knoten v zu zeigen, wird vorausgesetzt, dass die minimalen Gating-Pfade, welche an den Kinder-Knoten $w \in v.\mathrm{children}$ beginnen, zusammen mit ihrer Bedingung bekannt sind.

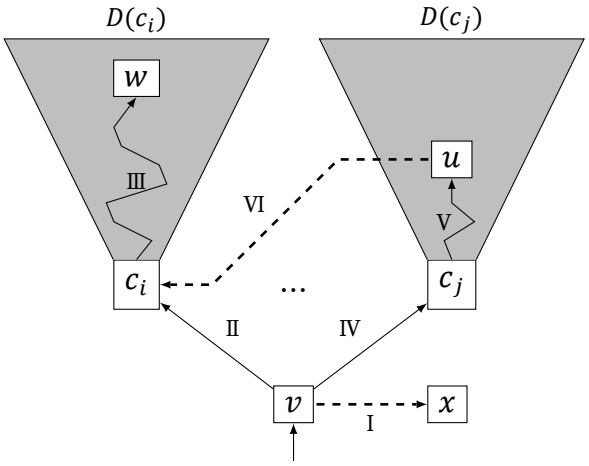


Abbildung 6.2: Szenarios für Gating-Bedingungen

Induktionsschritt

Für einen beliebigen Knoten v lassen sich die dort beginnenden Gating-Pfade in drei Kategorien unterteilen:

1. Gating-Pfade mit einer Länge von einer Kante.
2. Pfade $P = (v, c_i) \rightarrow (c_i, x_1) \rightarrow ... \rightarrow (x_n, w)$ mit $\{x_1, ..., x_n\} \subseteq D(c_i)$
3. Pfade $P = (v, c_j) \rightarrow (c_j, x_1) \rightarrow ... \rightarrow (x_n, w)$ mit $\{x_1, ..., x_n\} \nsubseteq D(c_j)$

Abbildung 6.2 dient zur Illustration dieser Fälle im Dominanzbaum. Durchgezogene Kanten entsprechen der Dominanzbeziehung. Daten-Kanten sind gestrichelt dargestellt.

Fall 1 Es handelt sich hierbei um dieselbe Situation wie beim Induktionsanfang. Die trivialerweise minimalen Gating-Pfade entsprechen den ausgehenden Kanten und die zugehörigen Bedingungen sind daher unmittelbar bekannt. Die Kante (I) ist ein Beispiel hierfür.

Fall 2 Wegen $\{x_1, ..., x_n\} \subseteq D(c_i)$ ist der im Kind-Knoten c_i beginnende Teilpfad $Q = (c_i, x_1) \rightarrow ... \rightarrow (x_n, w)$ selbst bereits ein Gating-Pfad von c_i und damit ist der Gesamtpfad P nicht minimal und muss nicht erfasst werden. Beispielsweise beinhaltet der Pfad (II+III) bereits den Gating-Pfad (III) von c_i und ist daher nicht minimal.

Fall 3 Der hierbei betrachtete Pfad P beinhaltet den Teilpfad $Q = (c_j, x_1) \rightarrow ... \rightarrow (x_n, w)$, der sich selbst nicht auf den Teilbaum $D(c_j)$ des Kindes c_j beschränkt. Diese Situation ist im Beispiel durch den Pfad (IV+V+VI+III) dargestellt. Interessant ist, dass die Kante (VI), welche zwischen vom Teilbaum $D(c_j)$ zum Teilbaum $D(c_i)$ wechselt, zwangsweise das Kind c_i zum Ziel haben muss.

Beweis. Nehmen wir an, es gäbe einen Knoten $b \neq c_i$ mit $b \in D(c_i)$ und eine Kante $e = (a, b)$ mit $a \in D(c_j)$. Die Dominanzbeziehung $b \in D(c_i)$ setzt voraus, dass ein Pfad von c_i nach b existiert. Dies gilt ebenso für c_j und a. Der zweite Pfad lässt sich durch die Kante e fortsetzen, so dass zwei Pfade zu b existieren müssen, wobei einer dieser Pfade durch c_j verläuft. Dies widerspricht der Annahme, dass b von c_i dominiert wird, denn $c_j \notin D(c_i)$. Es kann in diesem Fall also nur gelten: $b = c_i$. ■

Die Schlussfolgerung lautet, dass der Pfad in diesem Fall in mehrere Gating-Pfade zerfällt. Im Beispiel (6.2) lässt sich der Pfad so in die Gating-Pfade (IV), (V+VI) sowie (III) zerlegen. Per Induktionsannahme sind die an den Kind-Knoten beginnenden Gating-Pfade (V+VI) und (III) bereits bekannt. Der Teilpfad (IV) besteht aus nur einer Kante und ist gemäß Fall 1 einfach zu berechnen. Damit lassen sich die Bedingungen aller Teilpfade gemäß Definition 18 konjunktiv verketten.

Topologische Sortierung Das Beispiel zeigt einen Spezialfall, in dem nur zwei Teilbäume involviert sind. Tatsächlich kann der Pfad auch mehr als zwei Teilbäume umspannen. Sei $v \to A \to B \to C$ ein Pfad, der aus drei Gating-Pfaden besteht. Dabei muss zur Fortsetzung von $v \to A$ der Pfad $B \to C$ mit der zugehörigen Bedingung bekannt sein. Dies ist selbst *kein* Gating-Pfad, $v \to B \to C$ dagegen schon. Wir nennen einen solchen Pfad P daher *partiellen* Gating-Pfad mit einer zugehörigen Bedingung $\mathrm{gcp}(P)$ (analog zu gc).

Um eine wiederholte Berechnung zu vermeiden, ist es daher sinnvoll, Pfade wie $B \to C$ ebenso zu erfassen. Da der Graph (zumindest für die aktuellen Betrachtungen) azyklisch ist, lassen sich die Kinder von v topologisch sortieren, so dass die Berechnung mit dem Kind c_i beginnen kann, von welchem aus kein anderes Kind c_j erreichbar ist. Eine solche Sortierung kann garantieren, dass die partiellen Pfade $B \to C$ etc. bei Bedarf bekannt sind.

Verknüpfen der Bedingungen

Die Bedingungen aller minimalen Gating-Pfade P von v zu den Kindern c_i können nach der Verarbeitung von v zusammengefasst werden. Dies geschieht gemäß Definition 19.

6.1.5 Berechnungsalgorithmus

Der Algorithmus, welcher die Gating-Bedingungen für einen azyklischen Graphen berechnet, ist in Listing 7 dargestellt. Dabei erfolgt der erste Aufruf von CondsWalk mit dem `return`-Knoten als Argument.

Für den Algorithmus wird dabei angenommen, dass die Bedingungen gc und gcp zunächst undefiniert sind, also stets $\bot$ zurückliefern. Ebenso sind die Knoten anfangs nicht markiert. Kommt ein unbekannter Wert $\bot$ in einer Berechnung vor, so muss er als false interpretiert werden, so dass $\bot \vee A = A$ gilt.

Die CondsWalk(v)-Prozedur ermittelt die Gating-Bedingungen für alle Gating-Pfade die in v beginnen. Dies geschieht bei der Rekursion durch den Dominanzbaum *post-order*. Daher sind an dieser Stelle auch noch keine Knotenmarkierungen erforderlich. Der erste Schritt ist jeweils die Bestimmung der Gating-Bedingungen für die partiellen Gating-Pfade. Sie erfolgt für alle Kinder von v.

Verarbeitung der partiellen Pfade

Listing 6 Bestimmung der partiellen Gating-Bedingungen

```
1:  procedure PartCondsWalk(v, w)
2:      if v.isMarked then
3:          return
4:      end if
5:      v.isMarked ← true
6:
7:      for all x ∈ v.children : gc(w ⇒ x) ≠ ⊥ do
8:          PartCondsWalk(v, x)
9:
10:         for all c ∈ {gc, gcp}, y ∈ V \ D(x) : c(x ⇒ y) ≠ ⊥ do
11:             gcp(w ⇒ y) ← gcp(w ⇒ y) ∨ (gc(w ⇒ x) ∧ c(x ⇒ y))
12:         end for
13:     end for
14: end procedure
```

Die PartCondsWalk-Prozedur (siehe Listing 6) erhält als Argumente den aktuellen Knoten v, sowie eines der Kinder w und berechnet Bedingungen für alle von diesem Kind ausgehenden partiellen Gating-Pfade. Das heißt alle Pfade, die ein oder mehrere andere Kinder von v durchlaufen.

Dabei wird die in Abschnitt 6.1.4 angesprochene topologische Sortierung implizit mittels Tiefensuche und *post-order*-Berechnung durchgeführt. Hierfür sind auch die Knotenmarkierungen erforderlich, die jeweils in Zeile 5 gesetzt werden. Sie verhindern die wiederholte Berechnung bei der Rekursion in Zeile 8.

Das Vorgehen ist nun folgendes: durch die post-order-Verarbeitung in CondsWalk sind die (minimalen) Gating-Pfade und -Bedingungen der Kinder von v bereits bekannt. Gesucht sind nun solche Gating-Pfade, die in w beginnen und in einem anderen Kind x von v enden.

Wird mindestens ein solcher Pfad von w nach x gefunden (d. h. die Bedingung $\mathrm{gc}(w \Rightarrow x)$ ist definiert und daher nicht $\bot$), so lässt sich dieser um sämtliche Gating-Pfade in x verlängern. Dies trifft auch auf die partiellen Gating-Pfade in x zu, die aber erst nach der Rekursion in Zeile 8 bekannt sind.

Das heißt, immer wenn mindestens ein Pfad zwischen x und einem weiteren Knoten y existiert, also $\text{gc}(x \Rightarrow y) \neq \bot$ oder $\text{gcp}(x \Rightarrow y) \neq \bot$ gilt, dann lässt sich eine (partielle) Bedingung für den fortgesetzten Pfad von w über x nach y bestimmen. Dabei ist zu beachten, dass Pfade die bereits vollständig von x dominiert werden (d. h. $y \in D(x)$) *nicht* verlängert werden, denn dies würde die Minimalitätsbedingung verletzen.

Die Verkettung der Bedingungen selbst erfolgt nach den Definitionen 18 und 19. Das heißt, die Bedingungen $\text{gc}(w \Rightarrow x)$ und $\text{cnd}(x \Rightarrow y)$ der Teilpfade werden per Konjunktion verkettet und mit bereits bekannten und partiellen Gating-Bedingungen für $w \Rightarrow y$ disjunktiv verknüpft.

Verarbeitung der konventionellen Pfade

Listing 7 Bestimmung der Gating-Bedingungen

```
1:  procedure CondsWalk(v)
2:      for all w ∈ v.children do
3:          CondsWalk(w)
4:      end for
5:
6:      for all w ∈ v.children do
7:          PartCondsWalk(v, w)
8:      end for
9:
10:     for all e = (v, w) ∈ v.succEdges do
11:         gc(v ⇒ w) ← gc(v ⇒ w) ∨ gc(e)
12:     end for
13:
14:     for all w ∈ v.children do
15:         for all c ∈ {gc, gcp}, x ∈ V \ D(w) : c(w ⇒ x) ≠ ⊥ do
16:             gc(v ⇒ x) ← gc(v ⇒ x) ∨ (gc(v ⇒ w) ∧ c(w ⇒ x))
17:         end for
18:     end for
19: end procedure
```

Sind alle partiellen Gating-Pfade bekannt, können die eigentlich gesuchten Gating-Pfade berechnet werden. Dies geschieht wieder in der CondsWalk-Prozedur aus Listing 7. Trivial sind dabei die Gating-Bedingungen für die ausgehenden Kanten von v, die in Zeile 11 gemäß Definition 17 bestimmt werden und – bei mehreren Kanten – gegebenenfalls disjunktiv verknüpft werden.

Die verbleibenden Gating-Bedingungen werden gebildet, indem (möglicherweise partielle) Gating-Bedingungen der Kinder-Knoten w vervollständigt werden. Auch hier können aufgrund der Minimalitätsbedingung die Gating-Bedingungen für Pfade übersprungen werden,

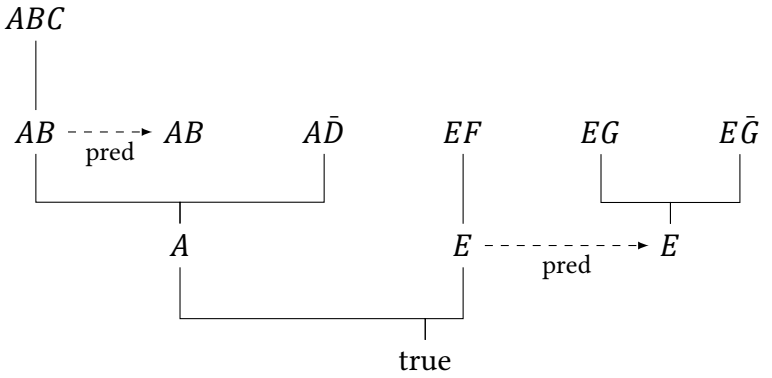


Abbildung 6.3: Regionen mit Baumstruktur

die bereits vollständig in $D(w)$ liegen. Nur bei Gating-Pfaden, welche (mit der letzten Kante) $D(w)$ verlassen, können weitere Pfade von v nach w entdeckt werden.

Die Umsetzung erfolgt ähnlich wie bei der Berechnung der partiellen Gating-Bedingungen. Es werden bekannte Gating-Bedingungen $\text{gc}(w \Rightarrow x) \neq \bot$ oder $\text{gcp}(w \Rightarrow x) \neq \bot$ für jedes Kind w gesucht. Diese werden mit $\text{gc}(v \Rightarrow w)$ verknüpft, um eine Bedingung für die Gating-Pfade von v über w nach x zu berechnen.

Zusammen mit den zuvor berechneten Kanten-Bedingungen decken diese alle minimalen Gating-Pfade aus den zuvor genannten drei Fällen ab.

6.2 Einteilung in Regionen

Die Berechnung der Gating-Bedingungen ermöglicht die Zusammenfassung der Knoten in Abhängigkeit von ihren Auswertungsbedingungen. Dabei müssen jedoch vorhandene Abhängigkeiten zwischen den Knoten berücksichtigt werden.

Sogenannte *Regionen* dienen dabei als Grundbausteine und entsprechen weitestgehend den späteren Grundblöcken. Sie sind jedoch in einer Baumstruktur organisiert und besitzen ebenso wie die individuellen Knoten eine Auswertungsbedingung. Für die Region R wird diese als R.cond bezeichnet. Ein Knoten lässt sich nur dann einer Region zuordnen, wenn er dieselbe Bedingung besitzt.

Abbildung 6.3 zeigt schematisch, wie die Regionen in eine Baumstruktur eingeordnet werden. Dabei wird jede Region durch ihre Bedingung dargestellt. Es gilt folgender Grundsatz: wenn eine Region R ein Kind S besitzt (in Anlehnung an die bisherige Notation: $S \in R$.children), dann muss die Bedingung S.cond die Bedingung R.cond implizieren, aber von ihr verschieden sein. Das heißt: die Auswertung von R ist *notwendig* für die Auswertung von S, nicht jedoch *hinreichend*. Je weiter man sich von der Wurzel des Baumes entfernt, desto spezifischer werden die Auswertungsbedingungen.

Dies ermöglicht es, die Knoten anhand ihrer Bedingung in einer Hierarchie zu organisieren, berücksichtigt jedoch noch keine Abhängigkeiten zwischen ihnen. Hierfür ist es notwendig, die Regionen mit einer Reihenfolge zu versehen.

Per Konvention gilt zunächst: Kinder werden *vor* ihren Eltern ausgewertet (dies erleichtert den Aufbau vom `return`-Knoten aus). Es ist hiermit alleine aber noch nicht möglich, mehrere Regionen mit derselben Bedingung zu ordnen.

Dies wäre beispielsweise in folgender Situation notwendig: Gegeben sind drei Knoten v, w und x, die über je eine Kante zu $v \to w \to x$ verbunden sind. Wenn v und x dieselbe Bedingung haben, w jedoch eine speziellere Bedingung aufweist, dann können die Knoten im CFG nicht demselben Grundblock zugeordnet werden. Dies würde entweder einen ungültigen CFG erzeugen oder man müsste w immer gemeinsam mit v und x auswerten, wodurch sich die Semantik ändern würde.

Um solche Abhängigkeiten zu berücksichtigen, können Regionen R eine Vorgänger-Region R.pred besitzen. Der spätere Aufbau des CFG garantiert, dass R.pred vor R und dem gesamten Teilbaum unterhalb von R ausgewertet wird. Beispielsweise sind in Abbildung 6.3 zwei Regionen AB vorhanden. Dies könnte daran liegen, dass zwei Knoten mit Bedingung AB über einen Knoten mit Bedingung ABC verbunden sind. Die Kind-Region ABC wird *vor* der Vater-Region AB ausgewertet, jedoch *nach* der Vorgänger-Region von AB.

6.2.1 Implikation von Knoten

Die Berechnung der Gating-Bedingungen erzeugt stets nur *relative* Bedingungen, zwischen Knoten v und unmittelbarem Dominator. Diese Unterteilung ist möglich, da die Auswertung des Dominators stets eine notwendige Bedingung ist. Die *absolute* Bedingung ergibt sich folglich, indem man alle relativen Bedingungen bis hin zur Wurzel des Dominanzbaumes konjunktiv verknüpft.

Um nun die Knoten einer bestimmten Region zuzuordnen, muss festgestellt werden, ob die absolute Bedingung der Region die Auswertung des jeweiligen Knoten impliziert. Dies kann nur dann der Fall sein, wenn alle relativen Teilbedingungen des Knotens impliziert werden.

Der Algorithmus aus Listing 8 prüft, ob eine solche Implikation vorliegt. Dabei wird zunächst die Teilbedingung bis zum nächsten Dominator geprüft (Zeile 5). Sobald eine Teilbedingung nicht durch die Bedingung der Region impliziert wird, kann abgebrochen werden.

Erst, wenn feststeht, dass die Teilbedingung impliziert wird, kann die nächste Teilbedingung – nämlich jene des Dominators – überprüft werden. Hierzu dient die Rekursion in Zeile 9. Wenn kein Dominator mehr vorhanden ist, dann wurden alle Teilbedingungen überprüft und es kann abgebrochen werden.

Listing 8 Überprüfen von Implikationen

```
 1: function RegionImplies(R, v)
 2:     d ← idom(v)
 3:     c ← (d ≠ ⊥) ? gc(d ⇒ v) : true
 4:
 5:     if ¬Implies(R.cond, c) then
 6:         return false
 7:     end if
 8:     if d ≠ ⊥ then
 9:         return RegionImplies(R, d)
10:     end if
11:
12:     return true
13: end function
```

Die Implies-Funktion überprüft dabei, ob eine Gating-Bedingung die andere impliziert. Dabei liegt die Bedingung der Region immer als einzelner Konjunktionsterm vor[1] und die Bedingung des Knotens üblicherweise in disjunktiver Normalform. Eine Implikation liegt dann vor, wenn ein Konjunktionsterm der disjunktiven Normalform vollständig im Konjunktionsterm der Bedingung enthalten ist. Für true und false sind gegebenenfalls Speziallfälle notwendig.

6.2.2 Aufbau des Regions-Baumes

Um einen Knoten einer Region zuzuordnen, muss bekannt sein, von welchen Regionen aus auf ihn zugegriffen wird. Dies folgt aus der Notwendigkeit, die Ausführung des Knotens vor seiner Verwendung zu gewährleisten. Die Kernidee des Algorithmus lässt sich daher wie folgt zusammenfassen:

1. Jeder Zugriff, der von einem Knoten v mit zugeordneter Region auf einen Knoten w erfolgt, liefert einen Hinweis für die Regionszuordnung von w.
2. Erst wenn alle eingehenden Kanten von v betrachtet wurden, reichen die Hinweise aus, um eine endgültige Region für v zu finden.

Die in Listing 9 dargestellte Prozedur implementiert dieses Kernprinzip. Sie wird auf den Kanten des Graphen ausgeführt und führt nur dann eine Rekursion zu einem neuen Knoten w durch, wenn tatsächlich alle Benutzer von w betrachtet wurden. Eine Rekursion liegt deshalb vor, weil PlaceSingle und PlaceMultiple indirekt wieder CreateRegions aufrufen.

Der Aufruf von AppendHint erzeugt dabei für jede betrachtete Kante e einen neuen Hinweis für w. Diese können gegebenenfalls von AppendHint kombiniert werden, so dass auch

[1] Eine konjunktive Verknüpfung von potentiell negierten Elementaraussagen.

bei mehreren Kanten nur ein Hinweis übrig bleibt. Abhängig von der Anzahl der Hinweise w.hints wird dann entweder PlaceSingle oder PlaceMultiple verwendet, um den Knoten w einer Region zuzuordnen.

Listing 9 Regionen erstellen und zuordnen

```
1:  procedure CreateRegions(R, e = (v, w))
2:      AppendHint(R, e)
3:
4:      if ∀u ∈ w.users : u.isMarked then
5:          if |w.hints| = 1 then
6:              PlaceSingle(w, w.hints_0)
7:          else
8:              assert |w.hints| > 1
9:              PlaceMultiple(w)
10:         end if
11:         w.isMarked ← true
12:     end if
13: end procedure
```

Hinweise erfassen

AppendHint in Listing 10 wird für jede eingehende Kante e des Knotens w aufgerufen. Zusätzlich wird der Prozedur die Region R übergeben, welche einen Hinweis für die Platzierung von w liefert und üblicherweise die Quellregion darstellt, aus welcher der Zugriff mit der Kante e stammt. Die Aufgabe besteht nun darin, eine geeignete Region *vor* R zu finden, die w aufnehmen kann.

Es stellt sich also die Frage, wie sich die Ausführungsbedingungen entlang einer Kante e ändern können. Offensichtlich ist, dass entlang von *true*- oder *false*-Kanten am γ-Knoten neue Bedingungen entstehen können. Außerdem können Bedingungen vereinfacht werden, wenn mehrere Zweigpfade von γ-Knoten zusammenlaufen. Da wir γ-Knoten später noch getrennt abhandeln werden, können wir für AppendHint annehmen, dass die Bedingung entweder identisch bleibt oder einfacher wird.

Vertikale Suche Die erste Aufgabe in AppendHint besteht also darin, den Regionsbaum von R aus aufwärts zu durchlaufen, um die höchste[2] Region zu finden, welche die Ausführung von w noch impliziert.

Dies geschieht mit Hilfe der ScanUp-Funktion, welche zuerst bis zur Wurzel des Baumes aufsteigt (Zeile 2) und dann beim Abstieg die erste Region bestimmt, welche die Ausführungsbedingung des Knoten impliziert (Zeile 7). Hierbei kommt auch die bereits vorgestellte

[2]Die Region, welche am nächsten an der Wurzel liegt und entsprechend die einfachste Bedingung besitzt.

Listing 10 Hinweise erfassen und vertikale Suche

```
 1: function ScanUp(R, v)
 2:     R_P ← (R.parent ≠ ⊥) ? ScanUp(R.parent, v) : ⊥
 3:
 4:     if R_P ≠ ⊥ then
 5:         return R_P
 6:     end if
 7:     if RegionImplies(R, v) then
 8:         return R
 9:     end if
10:
11:     return ⊥
12: end function
13:
14: procedure AppendHint(R, e = (v, w))
15:     R_S ← ScanUp(R, w)
16:
17:     if R_S ≠ R then
18:         R_S ← EnsurePred(R_S)
19:     end if
20:
21:     h_f ← ScanHints(R_S, w)
22:     h_F.edges ← h_F.edges ∪ {e}
23: end procedure
```

RegionImplies-Funktion zum Einsatz. In Anlehnung an Abbildung 6.3 wird dies als *vertikale Suche* bezeichnet, da man sich vertikal im Baum bewegt.

Der vertikale Aufstieg alleine garantiert zwar, dass die Bedingungen für w erfüllt sind, berücksichtigt aber nicht die Abhängigkeit zwischen v und w. Um dieser zu genügen, muss die Region R_S, in welcher w platziert wird, zumindest *vor* der Region R ausgewertet werden.

Dies ist allerdings nur dann notwendig, wenn die Region R tatsächlich verlassen wurde, also v und w unterschiedlichen Regionen angehören. Ist dies der Fall, dann wird in Zeile 18 mittels EnsurePred die Vorgänger-Region von R_S bestimmt oder – wenn nötig – erstellt. Per Definition liegt diese vor allen Kind-Regionen unterhalb von R_S und damit auch vor R.

Die resultierende Region dient als Ausgangspunkt der *horizontalen Suche*, welche in den Vorgänger-Regionen von R_S nach anderen Regions-Hinweisen für w sucht, die wiederverwendet werden können. Die Bezeichnung orientiert sich wieder an Abbildung 6.3, welche die Vorgänger-Regionen in einer horizontalen Reihe darstellt.

Horizontale Suche Die Vorgänger von R_S bilden eine *Kette* von Regionen, die – in Bezug auf die Kante e – alle in der Lage sind, den Knoten w aufzunehmen. Für die verketteten Regionen gibt es dabei zusätzliche Attribute, welche einen Bezug auf andere Regionen innerhalb der Kette ermöglichen. Die erste Region in dieser Kette ist über das Attribut R.base in jeder Region R der Kette zu finden. Für genau zwei Regionen A und B mit $B = A$.pred gilt beispielsweise: A.base $= B$.base $= A$.

Ob zwei Regionen zur selben Kette gehören, ist also aus einem Vergleich der *Basis-Region* ersichtlich. Zusätzlich besitzt jede Region einen Index R.index, der mit jeder neuen Vorgänger-Region erhöht wird. Dies ermöglicht es, die Position zweier Regionen innerhalb einer Kette zu vergleichen (da Regionen immer nur am Ende der Kette hinzugefügt werden). Wenn B also ein Vorgänger von A ist, dann gilt: $(A.\text{base} = B.\text{base}) \wedge (B.\text{index} > A.\text{index})$.

Da jede Region *vor* R_S in der Kette in der Lage ist, den Knoten w aufzunehmen, bietet es sich an, diesen Spielraum zu nutzen, um verschiedene Platzierungs-Hinweise zu kombinieren. Ist beispielsweise bereits ein Hinweis für einen Vorgänger von R_S vorhanden, so kann dieser wiederverwendet werden, um eine zweifache Platzierung des Knotens zu vermeiden. Diese horizontale Suche wird von der ScanHints-Funktion durchgeführt (Listing 12), welcher die Start-Region R_S, sowie der zu platzierende Knoten v übergeben werden.

Listing 11 Kombination mehrerer Hinweise

```
1:  a = ...                 // ①
2:  ...
3:  if P then
4:      x = a + 1
5:  end if
6:  ...                     // ②
7:  if Q then
8:      y = a - 1
9:  end if
10: ...                     // ③
```

Beispiel Das Beispiel in Listing 11 demonstriert einen solchen Fall, in dem die horizontale Suche zwei Hinweise kombinieren kann. Die mit ①, ② und ③ bezeichneten Abschnitte werden dabei durch Regionen repräsentiert, die zusammen eine Kette bilden. Für die beiden Verzweigungen gibt es entsprechende Kind-Regionen unterhalb von ③ und ②. Wenn nun die untere Verzweigung zuerst abgearbeitet wird, so erhält man als ersten Hinweis für die Platzierung von a die Vorgänger-Region von ③, nämlich ②.

Wie man am Beispiel sieht, ist es jedoch sinnvoller, a in Region ① zu platzieren. Dies kann beim Erkunden der ersten Verzweigung geschehen, die eine Platzierung in Region ① vorsieht. Da sich beide Hinweise auf Regionen derselben Kette beziehen, kann der Hinweis für Region ② eliminiert werden.

Listing 12 Horizontale Suche

```
1:  function ScanHints(R_S, v)
2:      h_F ← First({h ∈ v.hints | h.region.base = R_S.base} ∪ {⊥})
3:
4:      if h_F ≠ ⊥ then
5:          if R_S.index > h_F.region.index then
6:              h_F.region ← R_S
7:          end if
8:      else
9:          h_F ← new Hint(R_S)
10:         h_F.node ← v
11:         v.hints  ← v.hints ∪ {h_F}
12:     end if
13:
14:     return h_f
15: end function
```

Vorgehen In Zeile 2 wird unter den Regions-Hinweisen von v ein vorhandener Hinweis h für die Regions-Kette von R_S gesucht. Dieser ist an derselben Basis-Region h.region.base zu erkennen. Aufgrund der Konstruktion kann dabei höchstens ein Hinweis für jede Kette vorliegen.

Wird ein entsprechender Hinweis gefunden, dann dient die neuere Region (erkenntlich am größeren Index) als Ziel für den Knoten v. Die im Hinweis hinterlegte Region wird gegebenenfalls aktualisiert (siehe Zeile 6). Wenn dagegen noch kein Hinweis für eine Region der aktuellen Kette besteht, dann wird stattdessen ein neuer Hinweis erstellt und im Knoten hinterlegt (Zeile 9). Bei der Verarbeitung weiterer eingehender Kanten wird dieser Hinweis gegebenenfalls wiederverwendet.

Nach Abschluss der Funktion wird die Kante, über welche der aktuelle Knoten gefunden wurde, im selektierten Regions-Hinweis hinterlegt (Listing 10, Zeile 22). Diese Information ist erforderlich, wenn mehrere Regions-Hinweise aufgelöst werden müssen.

Mehrere Regions-Hinweise PlaceMultiple (Listing 13) wird aufgerufen, wenn alle Hinweise für den Knoten v gefunden wurden, jedoch nicht zu einem einzigen Hinweis reduziert werden konnten. In diesem Fall muss ein Duplikat des Knotens für jeden Hinweis erstellt werden.

Hierbei kommen die zuvor erfassten Kanten zum Einsatz. Sie geben an, welche Kanten für den jeweiligen Hinweis verantwortlich sind. Diese Kanten erzwingen die Auswertung des Knotens in der im Hinweis hinterlegten Region.

Damit ist das Vorgehen naheliegend: zunächst wird der Knoten für jeden Hinweis dupliziert. Dies wird hier durch die Funktion CopyNode(v) symbolisiert, welche die ausgehenden Kan-

Listing 13 Platzieren von Knoten in mehreren Regionen

```
 1: procedure PlaceMultiple(v)
 2:     h_0 ← First(v.hints)
 3:     h_0.node ← v
 4:
 5:     for all h ∈ v.hints \ {h_0} do
 6:         h.node ← CopyNode(v)
 7:
 8:         for all e ∈ h.edges do
 9:             e.dst ← h.node
10:         end for
11:     end for
12:
13:     for all h ∈ v.hints do
14:         PlaceSingle(h.node, R)
15:     end for
16: end procedure
```

ten v.depEdges mit kopiert. Die eingehenden Kanten sind jedoch nicht betroffen und müssen den Kopien neu zugeordnet werden. Dazu werden die dem Hinweis h zugeordneten Kanten h.edges auf die Kopie h.node umgeschrieben. Im Anschluss daran werden die einzelnen Kopien – wie alle anderen Knoten auch – mittels PlaceSingle platziert.

Eindeutige Regions-Hinweise Da die Region für den Knoten v nun eindeutig feststeht, erzeugt PlaceSingle (Listing 14) nun einfach die entsprechende Regions-Zuordnung (Zeile 2). Danach erfolgt die Rekursion zurück zur CreateRegions-Prozedur.

Für „normale“ Knoten erfolgt dabei eine einfache Rekursion entlang aller Abhängigkeitskanten (Zeile 12). Dies gibt dem Algorithmus die Gelegenheit, nun auch die Abhängigkeiten von v zu platzieren.

Ein γ-Knoten dagegen erfordert eine gesonderte Behandlung, da dieser neue Regionen unter R aufspannt. Dies geschieht durch die EnsureChildren-Funktion. Ähnlich wie EnsurePred erstellt die Funktion nicht immer neue Regionen. Stattdessen kann eine Region immer nur zwei Kind-Regionen je Bedingung besitzen: nämlich eine für true und false. Dabei überprüft EnsureChildren, ob entsprechende Regionen bereits existieren und gibt dann entweder die beiden vorhandenen Regionen für die Verzweigung zurück oder erstellt diese.

Außerdem wird mittels EnsurePred eine Vorgänger-Region für R erstellt, die als Hinweis für die Verzweigungsbedingung zum Einsatz kommt. Dies ist notwendig, um eine Auswertung der Bedingung *vor* der Verzweigung zu garantieren. Nachdem alle Regionen erstellt oder abgerufen wurden, erfolgt eine Rekursion entlang der entsprechenden γ-Kanten.

Listing 14 Platzieren von Knoten in einer Region

```
 1: procedure PlaceSingle(v, R)
 2:     v.region = R
 3:
 4:     if v.kind = γ then
 5:         R_P ← EnsurePred(R)
 6:         (R_T, R_F) ← EnsureChildren(R, v.cond)
 7:         CreateRegions(R_P, v.condEdge)
 8:         CreateRegions(R_T, v.trueEdge)
 9:         CreateRegions(R_F, v.falseEdge)
10:     else
11:         for all e ∈ v.depEdges do
12:             CreateRegions(R, e)
13:         end for
14:     end if
15: end procedure
```

6.3 Konstruktion des CFG

Mit dem Regions-Baum ist der Aufbau des CFG vergleichweise einfach, weil jede Region genau einem Grundblock entspricht. Dabei sind folgende Aufgaben zu erledigen:

1. Ketten von Regionen müssen in Ketten von CFG-Teilgraphen übersetzt und auf geeignete Weise verbunden werden.
2. Kind-Regionen müssen an den entsprechenden Stellen in die Kette integriert und mit entsprechenden Verzweigungen versehen werden.
3. Die Knoten müssen den jeweiligen Blöcken zugeordnet werden.
4. γ-Knoten müssen durch ϕ-Knoten ersetzt werden.

Der Algorithmus konstruiert den CFG vom Start-Block aus abwärts. Vorangehende Grundblöcke werden dabei von Funktion zu Funktion durchgereicht. Damit kann jede Funktion die erforderlichen (bedingten) Sprünge im Vorgänger-Block selbst erstellen und mit den neu konstruierten Grundblöcken verbinden. Zuerst wird dabei – mit dem Regions-Baum als Vorlage – ein Steuerflussgraph aus leeren Grundblöcken aufgebaut. Wenn dieses Gerüst steht, werden die Knoten in einem zweiten Durchlauf in die neuen Grundblöcke verschoben.

6.3.1 Regions-Ketten konstruieren

Die Konstruktion beginnt mit der Kette der „Wurzelregionen“ im Regions-Baum. Als erster Vorgänger-Block B_P bei der Konstruktion des CFG dient dabei ein leerer Grundblock, der direkt mit dem Start-Block verbunden ist.

Listing 15 Ketten von Regionen konstruieren

```
1:  function BuildChain(R, B_P)
2:      if R.pred ≠ ⊥ then
3:          B_P ← BuildChain(R.pred, B_P)
4:      end if
5:
6:      for all (R_T, R_F) ∈ R.children do
7:          B_P ← BuildChildren((R_T, R_F), B_P)
8:      end for
9:
10:     if |R.children| = 0 then
11:         J   ← new Jump(B_P)
12:         B_P ← new Block()
13:         Connect(J, B_P)
14:     end if
15:
16:     R.block ← B_P
17:     return B_P
18: end function
```

Die Konstruktion des CFG für solche Regions-Ketten ist für sich genommen relativ simpel (siehe Listing 15), weil ein Großteil der Arbeit mit auf die BuildChildren-Methode abgewälzt wird. Diese erzeugt nämlich Verzweigungen und Teilgraphen für die Kind-Regionen und gibt jeweils einen einzigen Grundblock zurück, in dem der Steuerfluss nach Verarbeitung der Verzweigung wieder zusammenläuft.

Das heißt: die Regionen der Kette werden der Reihe nach von Vorgänger zu Nachfolger abgearbeitet und miteinander verknüpft. Die Reihenfolge wird durch die Rekursion mit post-order-Verarbeitung gewährleistet.

Ausgehend von dem übergebenen Start-Block können nun die Kinder jeder Region in der Kette konstruiert werden (Zeile 7). Die Reihenfolge der Kind-Regionen spielt dabei keine Rolle. Der von BuildChildren zurückgegebene Grundblock kann dann anschließend als Start-Block für die nächste Kind-Region dienen.

Wurden alle Kinder abgearbeitet, so lässt sich der Grundblock B_P für R selbst verwenden (siehe die Zuweisung in Zeile 16). Dieser wird auch zurückgegeben, so dass ein Aufrufer am Ende der Verarbeitung den letzten Grundblock der Kette als Rückgabe erhält.

Nur wenn eine Region keine Kinder besitzt (Zeile 7 also nie aufgerufen wird), muss ein expliziter Blockwechsel erzwungen werden. Dazu dient die Konstruktion ab Zeile 11. Die Verbindung von Sprunginstruktion und Grundblock wird hier explizit durch die Connect-Prozedur hergestellt.

6.3.2 Kind-Regionen konstruieren

Listing 16 Ketten von Regionen konstruieren

```
1:  function BuildChildren((R_T, R_F), B_P)
2:      J ← new Cond(B_P, R.cond)
3:      B_T, B_F ← new Block(), new Block()
4:      Connect(J, B_T, kind = true)
5:      Connect(J, B_F, kind = false)
6:
7:      B_T ← BuildChain(R_T, B_T)
8:      B_F ← BuildChain(R_F, B_F)
9:
10:     J_T, J_F ← new Jump(B_T), new Jump(B_F)
11:     B_M      ← new Block()
12:     Connect(J_T, B_M)
13:     Connect(J_F, B_M)
14:
15:     return B_M
16: end function
```

Die Funktion erstellt die notwendige Verzweigung, um die Bedingungen der übergebenen Regionen sicherzustellen. Wie in Listing 15, Zeile 7 zu sehen, werden die Regionen stets paarweise übergeben. Dies ist sinnvoll, um beide Ergebnisse der Verzweigung abzudecken. Regionen werden auch stets paarweise erzeugt (siehe Listing 14, Zeile 6) und können bei der Erstellung auch miteinander verknüpft werden, um eine schnelle Zuordnung beider Regionen zu gewährleisten.

Ansonsten besteht die Hauptaufgabe von BuildChildren darin, eine Verzweigung mit zwei Folgeblöcken zu erstellen und im Anschluss wieder zusammenzuführen (siehe Listing 16). Ausgehend von diesen zwei Grundblöcken B_T und B_F können die Teilgraphen für die Regionen R_T und R_F wieder mit der BuildChain-Funktion konstruiert werden.

Die letzten Grundblöcke der entsprechenden Teilgraphen werden zurückgegeben und wieder in B_T und B_F gespeichert. Von diesen Grundblöcken aus werden Sprünge in einen neuen Grundblock B_M konstruiert, welcher den Steuerfluss wieder vereint und als Ergebnis zurückgeliefert wird.

6.3.3 Knoten in Zielblöcke verschieben

Die Zuordnung der Knoten zu den Grundblöcken ist einfach, da PlaceSingle jedem Knoten eine Region zuordnet und BuildChain jeder Region einen Grundblock. Es müssen also lediglich die Knoten v durchlaufen und in v.region.block verschoben werden.

Die einzige Ausnahme von dieser Regel stellen die γ-Knoten dar. Da diese später durch ϕ-Knoten ersetzt werden sollen, muss sichergestellt werden, dass diese in einem Grundblock platziert werden, welcher die passenden Vorgänger-Blöcke aufweist. Offensichtlich muss dies einer der in BuildChildren konstruierten Grundblöcke B_M sein, in denen der Steuerfluss nach einer Verzweigung wieder zusammenläuft.

Eine Zuordnung ist hier möglich, indem man beim Aufspannen neuer Regionen durch einen γ-Knoten (Listing 14, Zeile 6) diese Regionen im γ-Knoten hinterlegt. Wenn nun noch beim Abbau in BuildChildren der Grundblock B_M an den Regionen R_T und R_F hinterlegt wird, dann lässt sich B_M vom γ-Knoten aus erfragen.

6.3.4 Gating-Funktionen ersetzen

Dies könnte im Grunde genommen gemeinsam mit dem letzten Schritt erledigt werden, indem man γ-Knoten nicht nur verschiebt, sondern gleichzeitig durch ϕ-Knoten ersetzt. Die Reihenfolge der true- und false-Werte entspricht dabei der in BuildChildren verwendeten Reihenfolge beim Verknüpfen der Grundblöcke. Da aber später bei der Verarbeitung der Schleifen noch weitere Gating-Funktionen hinzukommen, ist es sinnvoller, diese gemeinsam nach der Verarbeitung des gesamten Graphen – inklusive Schleifen – zu ersetzen.

6.4 Abbau am Beispiel

Der Abbau soll an dem in Abbildung 6.4 dargestellten vFirm-Graphen demonstriert werden. Der Graph wurde so gewählt, dass zwei Sonderfälle auftreten, auf die an gegebener Stelle noch genauer eingegangen wird.

6.4.1 Bestimmung der Gating-Bedingungen

Die Gating-Bedingungen zwischen Knoten und ihren Dominatoren können in diesem einfachen Beispiel weitestgehend direkt abgelesen werden. Da im Kapitel 6.1 bereits Beispiele zur Berechnung genannt wurden, werden die Ergebnisse der Analyse hier direkt genannt:

$$\begin{aligned}
\mathrm{gc}(\texttt{return} \Rightarrow C) &= \text{true} & \mathrm{gc}(C \Rightarrow F) &= \text{true} & \mathrm{gc}(\gamma_3 \Rightarrow B) &= \text{true} \\
\mathrm{gc}(C \Rightarrow \gamma_1) &= \text{true} & \mathrm{gc}(\gamma_1 \Rightarrow D) &= A & \mathrm{gc}(F \Rightarrow \gamma_4) &= \text{true} \\
\mathrm{gc}(C \Rightarrow \gamma_2) &= \text{true} & \mathrm{gc}(\gamma_2 \Rightarrow E) &= A \vee \bar{A}\bar{B} & \mathrm{gc}(\gamma_4 \Rightarrow G) &= \bar{A} \\
\mathrm{gc}(C \Rightarrow A) &= \text{true} & \mathrm{gc}(\gamma_2 \Rightarrow \gamma_3) &= \bar{A} & \mathrm{gc}(\gamma_4 \Rightarrow \texttt{start}) &= \text{true}
\end{aligned}$$

Es ist bereits zu erkennen, dass der Knoten E aufgrund seiner Gating-Bedingung einer der beiden Spezialfälle ist.

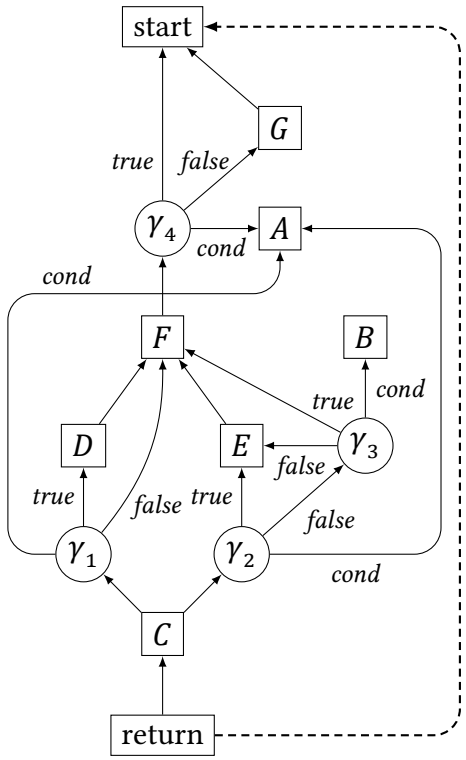


Abbildung 6.4: vFirm-Graph zum Beispiel

6.4.2 Aufbau der Regionen und Anordnung

Wurzelregion

Zum Vergleich ist der Regions-Baum mit den zugeordneten Knoten in Abbildung 6.5 dargestellt (Kanten zwischen Regionen fehlen aus Gründen der Übersichtlichkeit). Bei der Beschreibung des Ablaufes wird gegebenenfalls auf die einzelnen Regionen verwiesen.

`return` Der Algorithmus beginnt mit der Wurzelregion I und der Bedingung true. Zunächst wird der Region nur der `return`-Knoten zugeordnet. Ausgehend von diesem Knoten wird nun der Graph erkundet und Regions-Hinweise für die einzelnen Knoten gesucht. Nach CreateRegions erfolgt die Platzierung eines Knotens erst dann, wenn alle Kanten zu diesem Knoten erkundet wurden.

C, γ_1, γ_2 Dies besonders leicht, wenn nur eine einzige Kante zu einem Knoten existiert. In diesem Fall gibt der Benutzer-Knoten einen eindeutigen Regions-Hinweis an den betrachteten Knoten weiter. Abhängig von der Bedingung wird dann die im Hinweis erwähnte Region oder eine Vater-Region für diesen Knoten verwendet (durch ScanUp ermittelt). Da wir im Moment die Wurzelregion betrachten, ist ein Aufstieg ohnehin nicht möglich. Die Knoten C, γ_1 und γ_2 werden daher alle in der Wurzelregion platziert. Dies geschieht auch in Übereinstimmung mit den Bedingungen, die jeweils true sind und trivialerweise durch die Bedingung der Wurzelregion impliziert werden.

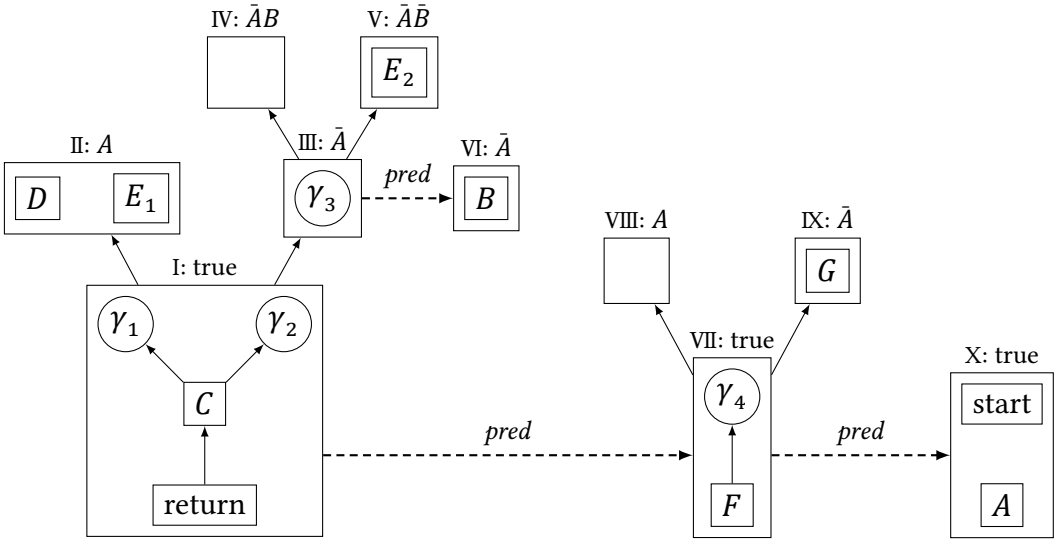


Abbildung 6.5: Regions-Baum zum Beispiel

Die erste Verzweigung

Bei der Verarbeitung der γ-Knoten werden außerdem weitere Regionen erstellt (siehe Listing 14, Zeile 6ff). Am offensichtlichsten sind dabei die beiden Kind-Regionen II und III, welche von γ_1 und γ_2 gemeinsam verwendet werden. EnsureChildren stellt sicher, dass Regionen mit derselben Bedingung nicht mehrfach erstellt werden.

Außerdem wird ein Vorgänger VII für die Wurzel-Region erstellt, welcher als Hinweis an die Bedingung A weitergegeben wird. Dies stellt sicher, dass die Bedingung vor der eigentlichen Verzweigung und den Kind-Regionen ausgewertet wird. Da aber noch nicht alle Kanten nach A bekannt sind, werden die beiden Hinweise lediglich abgespeichert. Da sie dieselbe Region andeuten, können sie trivialerweise kombiniert werden. ScanHints fügt keinen Hinweis hinzu, wenn $h_F \neq \bot$ (vgl. Listing 12).

Bei der Platzierung der γ-Knoten werden die neuen Regionen II und III als Hinweise für D und γ_3 an CreateRegions durchgereicht. Die Prüfung mittels ScanUp ergibt, dass die Knoten tatsächlich nicht in eine der Vater-Regionen (hier nur die Wurzelregion) geschoben werden können. Aus $\mathrm{gc}(\gamma_1 \Rightarrow D) = A$ und $\mathrm{gc}(\gamma_2 \Rightarrow \gamma_3) = \bar{A}$ ist nämlich leicht ersichtlich, dass die Verzweigung für beide Knoten von Relevanz ist. Da nun alle Kanten nach D und γ_3 betrachtet wurden, werden diese wieder mittels PlaceSingle in den jeweils eindeutigen Regionen II bzw. III platziert. Dabei erzeugt γ_3 zwei weitere Kind-Regionen unter III. D, γ_3

Wie bereits beim Knoten A wird für die Bedingung B von γ_3 eine Vorgänger-Region VI zu III erstellt und an B übergeben. Davon abgesehen ist interessant, dass die Prüfung der Bedingung mittels RegionImplies in ScanUp tatsächlich auch die Bedingungen des Dominators γ_3 überprüfen muss, um zu bestätigen, dass die Wurzelregion nicht für B geeignet ist. Zwar ist $\mathrm{gc}(\gamma_3 \Rightarrow B) = \mathrm{true}$, die absolute Bedingung ist jedoch $\mathrm{gc}(\texttt{return} \Rightarrow B) = \bar{A}$. B

Duplikation von Knoten (Sonderfall 1)

E Der Knoten E ist interessant. Von γ_2 liegt bereits ein Hinweis für die Region II vor, aber durch γ_3 kommt noch ein Hinweis für V dazu. AppendHint wird also versuchen, diese Hinweise zu kombinieren. Dies ist allerdings nur dann möglich, wenn beide Hinweise an zwei verschiedenen Stellen einer Kette von Vorgänger-Regionen vorliegen. Wie man in Abbildung 6.5 leicht sehen kann, ist dies bei den Regionen II und V nicht der Fall.

Eine Duplikation zu vermeiden, ist in diesem Fall nicht einfach, da der Knoten im Fall $\bar{A}B$ *nicht* ausgewertet werden soll. Dies verhindert eine Verschiebung des Knotens in die Wurzelregion. Die gleiche Situation wird auch in der Dissertation von Lawrence [Law07] betrachtet und dort als „*Exclusive Redundancy*“ bezeichnet. Leider ist die Übertragung dieses Verfahrens auf den hier verwendeten Abbau nicht trivial[3].

Da aber zur Laufzeit sowieso nur ein Pfad gewählt werden kann, findet auch beim Erstellen einer Kopie keine mehrfache Auswertung statt. Die Optimierung dieses Spezialfalles würde daher im Wesentlichen die Codegröße beeinflussen. Dies könnte zwar Auswirkungen auf die Cache-Lokalität haben, erscheint für eine erste Implementation des vFirm-Abbaus allerdings vertretbar.

Kurz gesagt: beide Hinweise bleiben bestehen und folgerichtig wird der Knoten auch mittels PlaceMultiple platziert. Hierbei erfolgt die Duplikation in die Knoten E_1 und E_2, die in Abbildung 6.5 dargestellt sind. Die Kanten, über welche die Hinweise eingegangen sind, werden auf die jeweilige Kopie umgeschrieben (vgl. Listing 13). Beide Kopien von E geben damit unterschiedliche Hinweise an F weiter.

F Diverse Kanten laufen im Knoten F zusammen. Wegen $\mathrm{gc}(\texttt{return} \Rightarrow F) = \mathrm{true}$ können diese durch ScanUp alle auf die Wurzelregion I reduziert werden. Da aber Zugriffe aus den Verzweigungen heraus erfolgen, muss F vor diesen Verzweigungen platziert werden. AppendHint stellt dies beim Festlegen der Region sicher (Listing 10, Zeile 18). Ansonsten erzeugen alle Kanten denselben Hinweis für Region VII, so dass die Platzierung mittels PlaceSingle eindeutig ist.

Unterbrochene Verzweigungen (Sonderfall 2)

γ_4 Die Platzierung von γ_4 erfolgt trivialerweise in derselben Region wie F, so dass Region VII zwei neue Kind-Regionen VIII und IX erhält. Dadurch entsteht eine zweite Verzweigung mit der Bedingung A.

Tatsächlich lassen sich die beiden Verzweigungen ohne eine Codeduplikation nicht zusammenfassen. Der Knoten F wirkt hierbei als Unterbrechung: einerseits wird aus der zweiten Verzweigung auf das unbedingte F zugegriffen und andererseits benötigt F das Ergebnis der

[3] Die Grundidee besteht darin, die Auswertung des Knotens beim Verlassen der Verzweigungen durchzuführen, indem man für beide Zweige A und $\bar{A}\bar{B}$ einen gemeinsamen „Rücksprung-Block“ verwendet.

ersten Verzweigung. Die Zusammenfassung beider Verzweigungen würde eine Duplikation von F erfordern.

Da eine solche Optimierung auch im Vorfeld auf dem vFirm-Graphen erfolgen kann (durch Duplikation von F und Kombination der γ-Knoten), ist es wenig sinnvoll, die Abbauphase zusätzlich hiermit zu belasten.

A

Mit der Verarbeitung von γ_4 ist die letzte Kante nach A bekannt geworden. Wie bei jeder Bedingung weist diese auf die Vorgänger-Region des γ-Knotens hin, in diesem Fall auf die Region X. Damit liegen zwei Hinweise vor, die in derselben Kette von Regionen liegen. Einmal für die Region VII und für die Region X. ScanHints identifiziert in dieser Situation die neuere Region X und verwendet diese für A (vgl. Listing 12). Damit können beide Verzweigungen auf dieselbe Bedingung zugreifen.

G, `start`

Die Zuordnung der letzten verbleibenden Knoten sollte nun auch klar sein. Der Knoten G behält aufgrund seiner spezifischen Bedingung die als Hinweis übergebene Region IX. Der `start`-Knoten erhält zwei Regions-Hinweise, steigt aber aufgrund der Bedingung bis zur Wurzel VII auf und wechselt dabei zur Vorgänger-Region X, um den Zugriff aus der Verzweigung zu gewährleisten. Damit sind alle Knoten im Beispiel einer Region zugeordnet. Abbildung 6.6 zeigt das Beispiel als fertig konstruierten CFG.

6.5 Erweiterung auf Schleifen

Der bisherige Algorithmus orientiert sich in seinen Grundzügen an der Arbeit von Lawrence [Law07]. Leider lässt sich dieser aber aufgrund der dort verwendeten anderen Darstellung nicht auf Schleifen erweitern.

Ein erster Ansatz war, die wiederholte Auswertung von Knoten ähnlich wie die bedingte Ausführung als Gating-Bedingung aufzufassen. Zusätzliche Schleifenregionen waren dann in der Lage, diese Bedingung zu erfüllen, so dass die Platzierung des Knotens in diesen Regionen möglich wurde.

Zur Berechnung der Bedingungen musste der Graph zuerst in eine azyklische Form gebracht werden, um die Terminierung des Algorithmus zu gewährleisten. Außerdem wurden die Gating-Bedingungen nicht zu aussagenlogischen Formeln verallgemeinert. Stattdessen gab es einen eingeschränkten Satz von Vereinfachungsregeln, der bewusst *keine* Kommutativität umfasste.

Wie sich im Nachhinein herausstellte, brachte dieser Ansatz einige Probleme mit sich. Der folgende Abschnitt betrachtet diese Probleme und zeigt, wie sich hieraus die Ideen für das überarbeitete Verfahren entwickelt haben.

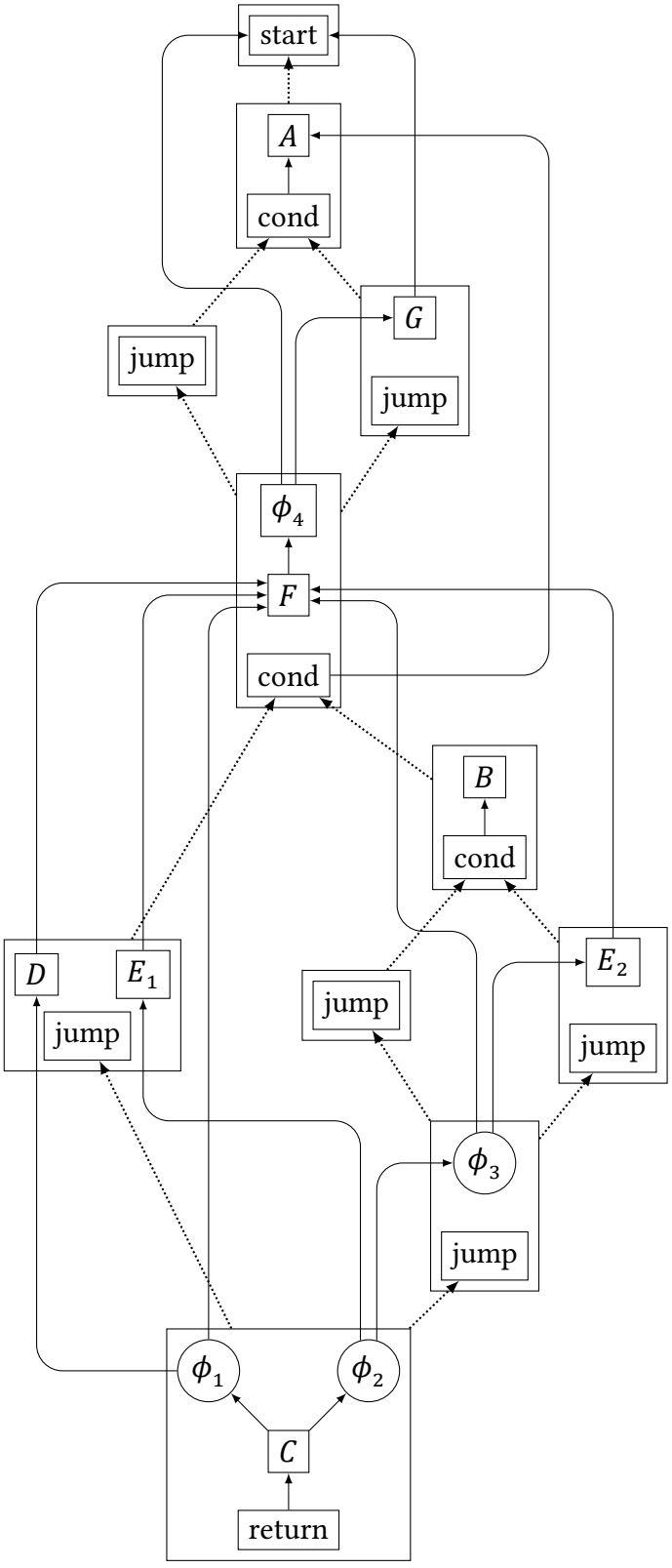


Abbildung 6.6: Vollständig konstruierter CFG

6.5.1 Konzeption

Gating-Pfade durch η-Knoten

In der Regel ist es nicht sinnvoll, Gating-Pfade zu betrachten, welche durch η-Knoten verlaufen, denn ähnlich wie bei den Dominatoren ist die Auswertung des η-Knotens immer notwendige Bedingung für die Ausführung eines Knotens innerhalb der Schleife. Ausgenommen hiervon sind nur Knoten, die mehreren Schleifen angehören und damit mehrere zugeordnete η-Knoten besitzen. Da ein Knoten in Firm jedoch stets nur einem Grundblock angehören kann, müssen solche Knoten ohnehin dupliziert werden.

Verwendet man dennoch solche Gating-Bedingungen, so kann dies dazu führen, dass sich Bedingungen inner- und außerhalb der Schleife vermischen. Diese Vermischung bringt diverse Probleme mit sich, erzeugt aber vor allem keine neuen Informationen. Da die Bedingung für die Auswertung der Schleife an sich bereits am η-Knoten abzulesen ist, muss diese nicht zusätzlich noch Teil der Knoten-Bedingung sein. Davon abgesehen erschwert dies die Interpretation der Bedingungen, sowie die Vereinfachung, insbesondere wenn mehrere Pfade mit verschiedenen η-Knoten auftreten.

Kombination von Schleifen

Es ist aber generell nicht sinnvoll, die Auswertungsbedingung eines Knotens v innerhalb einer Schleife zu bestimmen, wenn noch nicht abschließend geklärt ist, welche η-Knoten zu einer Schleife kombiniert werden können und welche nicht.

Der Grund dafür ist die Tatsache, dass ein Knoten von mehreren η-Knoten aus erreichbar sein kann. Wenn also alle Pfade zum Knoten v in einer Bedingung kombiniert werden, dann impliziert dies, dass alle η-Knoten, die v erreichen können, in einer Schleife zusammenfasst werden können. Ist dies nicht der Fall, dann muss der Knoten einerseits dupliziert werden (da er in Firm nur einer Schleife angehören kann) und andererseits ändert sich die Menge der Pfade zu den Duplikaten von v, so dass beide eine andere Bedingung erhalten können.

Schleifen als Teilgraphen

Sinnvoller ist es dagegen, die Schleifen als Teilgraphen zu betrachten, die wiederholt ausgewertet werden. Ähnlich wie bisher der `return`-Knoten, erhalten die θ-Knoten sowie der durch die η-Knoten ausgezeichnete Rückgabewert der Schleife eine feste Bedingung true, die sich auf jede Iteration der Schleife bezieht. θ-Knoten müssen gezwungenermaßen in jeder Iteration der Schleife einen neuen Wert selektieren und der Rückgabewert muss ebenso zur Verfügung stehen, wenn die Schleife vom Schleifenkopf aus verlassen wird.

Dies zeigt auch, dass die Selektion der η- und θ-Knoten, welche eine Schleife bilden können, für die Bestimmung der Bedingungen innerhalb dieser Schleife essentiell ist. Eine strikte Pipeline-Verarbeitung, welche die Bestimmung aller Gating-Bedingungen in einer Phase

durchführt, ist damit nicht möglich. Stattdessen ist es notwendig, den Graphen in Subgraphen oder Schichten aufzuteilen, die nach und nach abgearbeitet werden können.

Praktisch gesehen wird dies folgendermaßen umgesetzt: ausgehend von den η-Knoten wird eine einfache Schleifenanalyse durchgeführt, welche die Knoten in der Schleife bestimmt und Zugriffe aus der Schleife heraus erkennt. Im Anschluss daran wird die gesamte Schleife aus dem Graphen entkoppelt und durch einen `loop`-Knoten ersetzt, welcher die Schleife vertritt. Der resultierende Graph enthält weder η- noch θ-Knoten und ist folglich azyklisch.

Dies bedeutet: auf dem Graphen kann der bisherige Algorithmus angewendet werden, welcher das Konzept der Schleife nicht kennt. Damit ist die Verarbeitung der ersten Schleifentiefe möglich. Wenn alle Knoten platziert sind, dann kann entschieden werden, welche `loop`-Knoten miteinander kombiniert werden können. Dies bestimmt maßgeblich die Bedingungen der – zur Zeit entkoppelten – Knoten innerhalb der Schleife.

Wenn dies erfolgt ist, dann müssen die Schleifen einzeln verarbeitet werden, wobei wieder dasselbe Verfahren zum Einsatz kommt: verschachtelte Schleifen werden entkoppelt und der azyklische Abbau-Algorithmus kommt zum Einsatz. Dies wird so lange wiederholt, bis alle Schleifenebenen abgearbeitet sind.

6.5.2 Analogie zur Verzweigung

Schleifen und Verzweigungen weisen eine ähnliche Struktur auf, die sich bei der Verarbeitung ausnutzen lässt. Vergleichen wir dazu die Schleife und Verzweigung, die in Listing 17 dargestellt sind.

Listing 17 Verzweigung und Schleife

```
1: while A /*①*/ do
2:     // ②
3: end while
4: // ③
```

```
1: if A /*①*/ then
2:     // ②
3: end if
4: // ③
```

Der einzige wirkliche Unterschied besteht in Bezug auf den Wechsel von Abschnitt ② nach Abschnitt ③. Während der Steuerfluss bei der Verzweigung direkt zum Abschitt ③ wechselt, erfolgt bei der Schleife – abhängig von der Abbruchbedingung – der Rücksprung in den Schleifenkopf ①. Ist die Bedingung A nicht erfüllt, so wird die Ausführung in beiden Fällen direkt bei Abschnitt ③ fortgesetzt.

Da wir bereits einen existierenden Algorithmus haben, der mit Verzweigungen umgehen kann, ist es am einfachsten, diese Ähnlichkeit auszunutzen, indem man die Schleife mit (mehreren) η- und θ-Knoten in einen γ-Knoten transformiert. Bei der Konstruktion des CFG aus dem Regions-Baum kann dann einfach die Rücksprungkante umgeschrieben werden, um die fertige Schleife zu erhalten.

6.5.3 Das Puzzle zusammensetzen

Damit sind für einen (Teil-)Graphen G folgende Schritte notwendig.

1. Verschachtelte Schleifen in G erfassen und durch `loop`-Knoten ersetzen.
2. Abbau durch den bestehenden Algorithmus, `loop`-Knoten wenn möglich kombinieren.
3. Wenn vorhanden, G in den umschließenden Graphen reintegrieren.
4. Duplikation von Knoten um disjunkte Teilgraphen für die Schleifen zu erhalten.
5. Die Teilgraphen der Schleifen in G einzeln abarbeiten (zurück zu Schritt 1).

Dies wird so lange wiederholt, bis alle Schleifen abgearbeitet wurden. Listing 18 stellt das Vorgehen im Pseudocode dar. Die Erläuterung der einzelnen Schritte erfolgt in den folgenden Abschnitten.

Listing 18 Konstruktion des Graphen mit Schleifen

```
 1: procedure BuildGraphWithLoops(G)
 2:     L ← new Queue()
 3:     DetachEtas(G)
 4:     BuildGraph(G, L)
 5:
 6:     while L ≠ ∅ do
 7:         for all l ∈ L do
 8:             EnsureDisjoint(G, l)
 9:         end for
10:         Enqueue(L, ⊥)
11:
12:         while (l ← Dequeue(L)) ≠ ⊥ do
13:             BuildLoop(l, L)
14:         end while
15:         Dequeue(L)
16:     end while
17: end procedure
```

6.5.4 `loop`-Knoten konstruieren

Der Aufbau der `loop`-Knoten, um die im Graphen vorkommenden Schleifen zu repräsentieren, muss nicht unbedingt schrittweise geschehen. Es ist auch möglich, vor dem Abbau gleich *alle* Schleifen zu ersetzen. Dargestellt wird dies hier durch die DetachEtas-Prozedur, welche ausgehend von den η-Knoten im Graphen alle Schleifen aus dem Graphen *entkoppelt*.

Die Verarbeitung der η-Knoten erfordert hierbei eine einfache Schleifenanalyse, um schleifeninvariante Werte zu erkennen und das Ausmaß der Schleife zu bestimmen. Wenn die η-Knoten durch `loop`-Knoten ersetzt werden, sind die schleifeninternen Knoten nicht mehr direkt erreichbar. Dies trifft möglicherweise auch auf schleifeninvariante Werte zu, die jedoch *vor* der Schleife ausgewertet werden müssen, damit bei der späteren Integration in den CFG keine ungültigen Kanten entstehen. Die schleifeninvarianten Knoten müssen daher in Form von Abhängigkeiten erfasst werden.

Ähnlich wie bei den Gating-Bedingungen können Informationen über Schleifen durch die Duplikation von Knoten verändert werden. Da die Anpassung dieser Informationen beim Duplizieren nicht immer einfach ist, ist es sinnvoller, eine lokal beschränkte Schleifenanalyse durchzuführen, wenn eine neue Schleife untersucht wird. Ausgenommen hiervon ist die Schleifentiefe, die sich beim Duplizieren einfach auf das Duplikat übertragen lässt. Da sie Grundlage für die Erkennung der Schleifengrenzen ist, bietet sich eine Vorberechnung an.

Schleifentiefe

Die Berechnung der Schleifentiefe erfolgt nach Definition 13 (Seite 30) und lässt sich in einem Durchlauf durch den Graphen bestimmen. Grund hierfür ist die Tatsache, dass Zyklen im Graphen immer bei einem θ-Graphen beginnen und diese eine festgelegte Schleifentiefe besitzen. Damit entstehen bei der Berechnung auch keine Zyklen, so dass eine Fixpunktiteration nicht notwendig ist. Das Prinzip der Schleifentiefe wurde bereits bei der Einführung von vFirm mit Beispielen beschrieben, so dass wir an dieser Stelle die Schleifentiefe gemäß Definition 13 als gegeben betrachten.

Schleifenknoten und -Grenzen

Mit Hilfe der Schleifentiefe ist die Erkennung von schleifeninvarianten Knoten und die Erfassung der Schleifengrenzen trivial. Für einen η-Knoten v müssen ausgehend von v.cond und v.value alle Abhängigkeiten erfasst werden, deren Schleifentiefe mindestens v.depth + 1 beträgt. Sobald ein Knoten diese Bedingung verletzt, ist er schleifeninvariant und muss nicht weiter betrachtet werden. Formal definiert sind dies Knoten mit:

$$\begin{aligned}\text{inv}(v) = \{w \mid &\exists \text{Pfad } (P = v \to x_1 \to \ldots \to x_n \to w) \wedge \\ &(\forall i \in \{1, \ldots, n\} : x_i.\text{depth} > v.\text{depth}) \wedge \\ &(w.\text{depth} \leq v.\text{depth})\}\end{aligned}$$

Basisgraph

Nach dem Ersetzen der η-Knoten durch DetachEtas und dem damit verbundenen entkoppeln der Schleifen, erhält man einen azyklischen Graphen auf Schleifentiefe 0. Damit kann der Graph unmittelbar durch BuildGraph zu einem CFG-Fragment aufgebaut werden (siehe

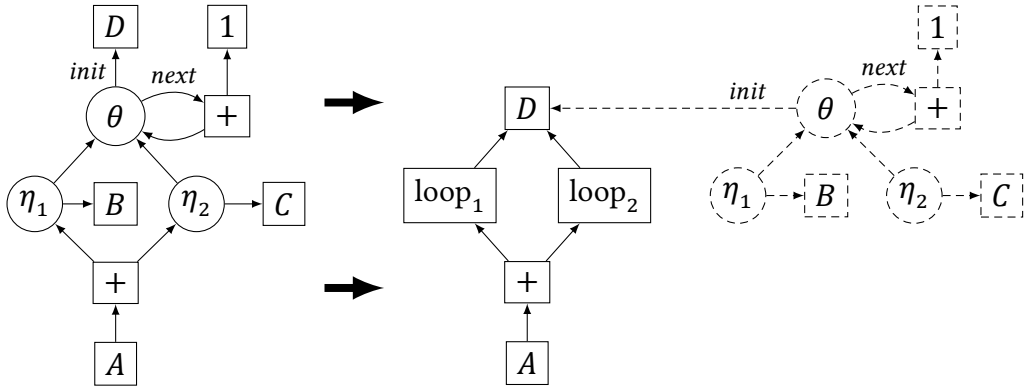


Abbildung 6.7: Entkoppeln von Schleifen mittels `loop`-Knoten

Listing 18). Dies umfasst die Konstruktion des Regions-Baumes mittels CreateRegions und die Konstruktion eines entsprechenden CFG mittels BuildChain.

Abbildung 6.7 demonstriert dies an einem einfachen Beispiel. Nach der Transformation erhält man einen azyklischen Basisgraphen mit `loop`-Knoten, während das Schleifenfragment ausgekoppelt wird und – in diesem Fall – vom Wurzelknoten A aus nicht mehr zu erreichen ist. Damit lässt sich BuildCFG anwenden.

6.5.5 Änderung am Abbaualgorithmus

Damit Schleifen später in den CFG integriert werden können, müssen einige Anpassungen an dem Aufbau-Algorithmus vorgenommen werden.

Hierfür wird die PlaceSingle-Prozedur (Listing 14) um zwei Sonderfälle erweitert. Der erste Sonderfall betrifft die Tatsache, dass `loop`-Knoten die Region, in der sie liegen, auftrennen müssen. Betrachten wir dazu das Beispiel aus Abbildung 6.7. Würden wir denselben Algorithmus wie bisher verwenden, dann könnte der gesamte (Basis-)Graph von A bis D in eine einzige Region eingeordnet werden, da keine γ-Knoten vorhanden sind. Im späteren CFG wird hieraus allerdings ein einziger Grundblock und dies nimmt uns die Möglichkeit, die Teilgraphen für die Schleifen in den resultierenden CFG einzubetten.

Eine solche Auftrennung der Grundblöcke kann gewährleistet werden, indem der `loop`-Knoten entlang seiner Abhängigkeitskanten einen Regionswechsel zur Vorgänger-Region erzwingt (ähnlich der Bedingungskante beim γ-Knoten).

Der zweite Sonderfall betrifft die Erfassung und Kombination von `loop`-Knoten. Zunächst einmal können diese bequem während ihrer Platzierung erfasst werden. Die in Listing 18 verwendete Warteschlange L dient diesem Zweck und wird entsprechend an BuildGraph weitergegeben. Nach dem Abbau des Basis-Graphen werden die in L gesammelten Schleifen abgearbeitet (wobei beim Abbau der Schleifen wiederum verschachtelte Schleifen zu L hinzukommen können).

Darüber hinaus ermöglicht die Platzierung der `loop`-Knoten die Kombination verschiedener Schleifen, solange diese dieselbe Abbruchbedingung besitzen und derselben Region zugeordnet werden können. Eine entsprechende Überprüfung in PlaceSingle ist trivial, wenn die `loop`-Knoten in einer Region erfasst werden. Die Kombination von Schleifen äußert sich zunächst nur darin, dass die involvierten `loop`-Knoten auf einander verweisen.

Da sich die Knoten innerhalb einer Kette von Regionen in gewissen Grenzen verschieben lassen, besteht unter Umständen noch das Potential, `loop`-Knoten in dieselbe Region zu verschieben und zu kombinieren. Andererseits verschieben sich hierdurch auch die Abhängigkeiten der `loop`-Knoten, so dass die Auswirkungen genau beachtet werden müssen. Eine entsprechende Optimierung wurde aus Zeitgründen bisher nicht umgesetzt, könnte aber unnötige Schleifen vermeiden.

Ein letzter Punkt ist beim Abbau noch zu beachten: für die Verarbeitung der Schleifen muss der Algorithmus auch auf diese beschränkt bleiben. Der Abbau erfolgt in Bezug auf die Schleifentiefe von unten nach oben. Daher können wir davon ausgehen, dass die Knoten, welche die Schleife umschließen, bereits verarbeitet wurden und nicht mehr innerhalb des alten vFirm-Grundblockes liegen. Es ist also am einfachsten, die Verarbeitung abzubrechen, wenn die betrachteten Knoten den vFirm-Grundblock verlassen.

6.5.6 Duplikation bei Schleifen

Bis zu diesem Zeitpunkt haben wir den CFG der Schleifentiefe 0 aufgebaut und alle dort vorkommenden `loop`-Knoten erfasst, sowie gegebenenfalls kombiniert. Damit kann jetzt die Verarbeitung der eigentlichen Schleifen beginnen. Diese ist in Listing 18 als Fixpunktiteration implementiert, wobei jede Iteration mit einer Duplikationsphase beginnt.

Die vFirm-Darstellung begünstigt die Eliminierung redundanter Knotenstrukturen durch Optimierungen wie die *Common Subexpression Elimination*. Viele elementare Strukturen, wie beispielsweise Schleifenzähler, werden daher von verschiedenen Schleifen geteilt.

Da in Firm jeder Knoten *höchstens einem* Grundblock angehören kann, ist es keine Seltenheit, dass Knoten wieder für die Verwendung in Schleifen dupliziert werden müssen. Dies ist aber erst nach der Erfassung und Kombination der `loop`-Knoten im letzten Schritt möglich. Dabei gilt: Schleifen, die zusammengefasst werden, können auf dieselben Knoten zugreifen, aber getrennte Schleifen müssen eine disjunkte Menge von Knoten verwenden.

Dies liefert die Grundidee für den Algorithmus: ausgehend von den beim Abbau erfassten `loop`-Knoten können die zur Schleife gehörigen Knoten markiert werden. Kombinierte `loop`-Knoten erhalten dabei dieselbe Markierung. Ist ein Knoten bereits mit einer anderen Markierung beschriftet, so gehört er zwei Schleifen an und muss dupliziert werden.

Der Algorithmus für diese Duplikation ist in Listing 19 dargestellt. Er wird für jeden gefundenen `loop`-Knoten l aufgerufen. Die als l.loopGroup bezeichnete Menge umfasst dabei alle mit l kombinierten `loop`-Knoten, sowie l selber. Alle kombinierten `loop`-Knoten werden im Anschluss durch EnsureDisjointWalk getrennt verarbeitet, wobei $G = l$.loopGroup selbst

Listing 19 Erzeuge disjunkte Schleifen-Teilgraphen

```
1:  procedure EnsureDisjoint(G = (V, E), l)
2:      G ← l.loopGroup
3:
4:      for all l_G ∈ G do
5:          for all d ∈ l_G.deps do
6:              d.isMarked ← true
7:          end for
8:          EnsureDisjointWalk(l_G.eta, G)
9:      end for
10: end procedure
11:
12: procedure EnsureDisjointWalk(v, M)
13:     if v.isMarked then
14:         return
15:     end if
16:     v.isMarked ← true
17:
18:     for all e ∈ v.depEdges do
19:         e.dst ← MarkOrClone(e.dst, M)
20:         EnsureDisjointWalk(e.dst, M)
21:     end for
22: end procedure
23:
24: function MarkOrClone(v, M)
25:     if v.loopMarker = M then
26:         return v.loopCopy
27:     end if
28:
29:     if v.loopMarker = ⊥ then
30:         v_L ← v
31:     else if v.loopMarker ≠ M then
32:         v_L ← CopyNode(v)
33:         v_L.loopCopy   ← v_L
34:         v_L.loopMarker ← M
35:     end if
36:
37:     v.loopCopy   ← v_L
38:     v.loopMarker ← M
39:     return v_L
40: end function
```

als Markierung für Knoten in der Schleife dient. Für die Markierungen v.loopMarker wird dabei ein Standardwert von $\perp$ angenommen. Für v.isMarked wie gewöhnlich false.

Um die Verarbeitung auf die eigentliche Schleife zu begrenzen, werden zuvor die schleifeninvarianten Knoten l_G.deps markiert, so dass die Verarbeitung an diesen Stellen abbrechen kann. Da der eigentliche Schleifengraph entkoppelt wurde, muss per l_G.eta auf den alten η-Knoten zugegriffen werden, welcher im eigentlichen Graphen nicht mehr erreichbar ist (vgl. Abbildung 6.7).

EnsureDisjointWalk führt nun eine einfache Tiefensuche auf dem Graphen durch, wobei der Zielknoten jeder Kante $e \in v$.depEdges mittels MarkOrClone neu aufgelöst und anschließend umgeschrieben wird. Wenn keine Duplikation stattfindet, wird schlicht e.dst zurückgeliefert, so dass keine Änderung stattfindet.

Damit richtet sich das Hauptaugenmerk auf die MarkOrClone-Funktion. Diese überprüft die Schleifenmarkierung v.loopMarker des Knotens und entscheidet je nach deren Wert das weitere Vorgehen. Existiert noch keine Markierung ($= \perp$), so wird der Knoten selbst der Schleife zugewiesen. Ist er stattdessen bereits für eine andere Schleife reserviert ($\neq M$), dann wird stattdessen eine Kopie erstellt.

Interessant ist dabei das Zusammenspiel von v.loopMarker und v.loopCopy. Ersteres dient weniger dazu, den Knoten einer Schleife zuzuordnen, sondern vermerkt ob der Knoten für die aktuelle Schleife bereits bearbeitet wurde. Wenn dies der Fall ist, dann verweist v.loopCopy auf die Kopie des Knotens, welche für die aktuelle Schleife erstellt wurde. Dies vermeidet die Erstellung mehrerer Kopien, wenn ein Knoten über verschiedene Kanten erreicht werden kann. Wenn keine Kopie erforderlich ist, dann verweist v.loopCopy auf den Knoten selbst.

Durch diesen Mechanismus wird auch die Abkürzung im Fall v.loopMarker $= M$ ermöglicht, welche einfach den M zugewiesenen Knoten v.loopCopy zurückliefert.

6.5.7 Abbau der Schleifen

Vor dem Aufbau der Schleifen wird eine Markierung zur Warteschlange hinzugefügt (Listing 18, Zeile 10). Sie trennt die `loop`-Knoten unterschiedlicher Schleifentiefe voneinander und ermöglicht so eine systematische Konstruktion, die schichtenweise vorgeht.

Das Herzstück des Aufbaus liefert die BuildLoop-Prozedur. Sie ist in Listing 20 dargestellt und reduziert den Aufbau der Schleife, wie zuvor beschrieben, auf den Aufbau einer Verzweigung. Zu diesem Zweck müssen zunächst per Schleifenanalyse für die Konstruktion relevante Knoten bestimmt werden (FindLoopNodes, Zeile 2). Dies sind die Abbruchbedingung c, die durch η-Knoten selektierten Ergebnisse R, sowie die θ-Knoten Θ.

Die Bestimmung dieser Knoten verläuft ähnlich wie die Bestimmung der Schleifeninvarianten zuvor. Das heißt, ausgehend von den η-Knoten (die bei den `loop`-Knoten hinterlegt sind), werden die Knoten der Schleife durchlaufen bis ein invarianter Knoten gefunden wird. Auf

Listing 20 Aufbau einer Schleife

```
1:  procedure BuildLoop(l, L)
2:      (c, R, Θ) ← FindLoopNodes(l)
3:      T_H ← new Tuple({c} ∪ R ∪ Θ) :
4:      T_B ← new Tuple({v.next | v ∈ Θ} ∪ {T_H})
5:      r   ← new γ(c, T_H, T_B)
6:
7:      R   ← BuildRegions(r)
8:      R_P ← R.pred
9:      (R_T, R_F) ← First(R.children)
10:
11:     B_H^in  ← NodeCopy(l.block)
12:     B_H^out ← BuildChain(R_P, B_H^in, L)
13:
14:     J   ← new Cond(B_H^out, c)
15:     B_B^in ← new Block()
16:
17:     l.block.preds ← ∅
18:     Connect(J, l.block, kind = true)
19:     Connect(J, B_B^in, kind = false)
20:
21:     B_B^out ← BuildChain(R_F, B_B^in, L)
22:     Connect(B_B^out, B_H^in)
23: end procedure
```

dem Weg dorthin werden interessante Knoten erfasst. Tiefere Schleifen sollten bereits entkoppelt sein, daher werden Knoten in den verschachtelten Schleifen wie gewünscht ignoriert. Diese gefundenen Knoten bilden die Wurzelpunkte für die Teilgraphen von Schleifenkopf und -körper.

Dabei bilden die Ergebniswerte der η-Knoten, Schleifenabbruchbedingung und θ-Knoten die Wurzel für den Teilgraphen des Schleifenkopfes. Sie müssen immer im Schleifenkopf zur Verfügung stehen. Im Falle der Abbruchbedingung ist dies offensichtlich, da der bedingte Sprung am Ende des Schleifenkopfes davon abhängt. θ-Knoten müssen ebenso in jeder Iteration der Schleife einen neuen Wert bestimmen und da ein Abbruch jederzeit möglich ist, muss das Ergebnis der Schleife ebenso feststehen.

Als Wurzel für den Schleifenkörper dienen die *next*-Kanten der θ-Knoten. Ihre Auswertung erfolgt nur dann, wenn der Schleifenkopf durch einen Rücksprung erneut betreten wird oder anders gesagt: wenn die Abbruchbedingung nicht erfüllt ist.

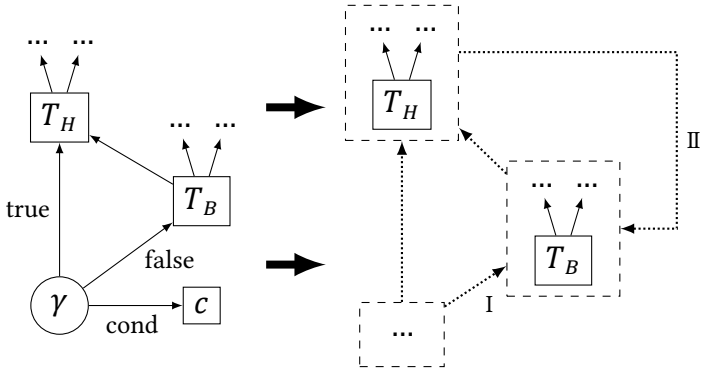


Abbildung 6.8: Hilfskonstruktion für Schleifen

Konstruktion der Verzweigung

Aus den Wurzelknoten werden nun zwei Tupel konstruiert, wobei die Wurzel des Schleifenkörpers – dargestellt durch das Tupel T_B – an die Abbruchbedingung geknüpft werden soll. Hierzu dient der in Zeile 5 konstruierte γ-Knoten.

Nur wenn die Abbruchbedingung verletzt wird, darf die Auswertung von T_B erfolgen. Entsprechend wird T_B bei der Konstruktion des γ-Knotens als false-Wert verwendet. Um T_H unabhängig von der Bedingung auszuwerten, muss das Tupel von beiden Ästen der Verzweigung aus abgerufen werden. Zu diesem Zweck enthält T_B einen Querverweis auf T_H.

Abbildung 6.8 zeigt die Konstruktion mit ihrer Übersetzung in einen CFG schematisch. Die gestrichelten Blöcke stehen dabei exemplarisch für ganze Teilgraphen, die aus mehreren Grundblöcken bestehen können. Man kann leicht erkennen, wie alleine durch das Umschreiben der Kante I nach II eine Schleife entsteht.

Ist der γ-Knoten r erstellt, dann wird die bereits bekannte Konstruktion des Regions-Baumes mittels BuildRegions angestoßen (die Funktion erstellt eine Wurzelregion und ruft dann CreateRegions auf). Im Anschluß lassen sich die Wurzel-Regionen für Schleifenkopf und -körper leicht bestimmen, da der Regions-Baum durch die Konstruktion stets dieselbe Grundstruktur besitzen muss. R_P als Vorgänger der Verzweigung ist die Wurzel des Schleifenkopfes, R_F entsprechend die Wurzel des Schleifenkörpers.

Mit diesen Teilbäumen im Regions-Baum lassen sich die CFG-Fragmente für die Schleife aufbauen und im Graphen integrieren. Hierfür wird die Schleife vor dem Grundblock eingebaut, welcher die `loop`-Knoten der Schleife enthält. Eine Kopie B_H^{in} dieses Grundblockes dient als Einstiegspunkt für die Schleife und wird später als Ziel für den Rücksprung und als Container für die θ-Knoten verwendet. Entsprechend der NodeCopy-Semantik werden die Vorgänger-Blöcke übernommen. Dies verbindet die Schleife mit dem restlichen Graphen.

Um die θ-Knoten später durch ϕ-Knoten ersetzen zu können, sind genau zwei Vorgänger-Blöcke notwendig. Der CFG-Aufbau muss dies sicherstellen, indem er garantiert, dass Grundblöcke mit `loop`-Knoten immer nur einen Vorgänger-Block besitzen (der zweite Vorgänger

wird in Zeile 22 hinzugefügt). Dies ist aber einfach möglich, indem ggf. weitere Grundblöcke erzeugt werden.

Von B_H^{in} aus wird nun der Schleifenkopf aufgebaut (Zeile 12), dessen CFG in B_H^{out} endet. Die BuildChain-Funktion wurde hierfür um den Parameter L erweitert, um `loop`-Knoten in der Schleife zu erfassen.

Danach erfolgt die Konstruktion der bedingten Verzweigung, welche entweder zum zuvor kopierten Grundblock l.block springt, um die Schleife abzubrechen oder einen neuen Grundblock B_B^{in} ansteuert, welcher als Startpunkt für den Schleifenkörper dienen soll. Die alten Vorgänger von l.block, welche B_H^{in} übernommen hat, werden dabei verworfen.

Für B_B^{in} erfolgt dann die Konstruktion des Schleifenkörpers, mit dem End-Block B_B^{out}, welcher im Anschluß mit dem Start-Block B_H^{in} des Schleifenkopfes verbunden wird.

Abbau der Gating-Knoten

Der Abbau der Gating-Knoten ist im Anschluß einfach. Kanten zu `loop`-Knoten können durch eine Kante zum Ergebnis des entsprechenden η-Knotens ersetzt werden, wodurch der Schleifen-Teilgraph wieder in den eigentlichen Graphen reintegriert wird.

Die θ-Knoten können einfach durch einen entsprechenden ϕ-Knoten mit zwei Eingängen ersetzt werden, müssen allerdings auch in den ersten Grundblock B_H^{in} der Schleife verschoben werden. γ-Knoten können ebenso durch entsprechende ϕ-Knoten ersetzt werden (siehe Abschnitt 6.3.4).

7 Ergebnisse

Im Rahmen dieser Arbeit ist eine Implementierung von vFirm mit dem hier beschriebenen Auf- und Abbau entstanden, die etwa 5000 Codezeilen umfasst. Obwohl die Implementierung des Abbaualgorithmus noch ausbaufähig ist, wurde die Transformation bereits für einige Testprogramme erfolgreich getestet.

Dafür wurden die Testprogamme mit dem für Firm entwickelten CParser-Compiler [IPD] in die Firm-Darstellung übersetzt und dann durch die Aufbauphase in vFirm transformiert. Um den Maschinencode zu erzeugen, wurde wieder der Abbau mit dem Firm-Backend gekoppelt. Hierdurch wird die Transformation Firm ↔ vFirm in beiden Richtungen getestet und einfache Betrachtungen zur Qualität der reinen Transformation sind möglich.

7.1 Testprogramme

Einfache Tests wurden zunächst durchgeführt, um die korrekte Umsetzung der elementaren Steuerfluss-Strukturen zu testen. Die wichtigsten sind hier genannt:

Name	Beschreibung
`recn_fibonacci`	Berechnung der Fibonacci-Folge[1] per Rekursion.
`loop_fibonacci`	Variante unter Verwendung einer Schleife.
`recn_ext_euclidean`	Erweiterter Euklidischer Algorithmus[2] per Rekursion.
`loop_ext_euclidean`	Variante unter Verwendung einer Schleife.
`spcl_branch`	Test der in Abschnitt 6.4 betrachteten Sonderfälle.
`spcl_nested_loop`	Berechnungen mit verschachtelten Schleifen.
`spcl_switch`	Durch `switch` hervorgerufene stark verzweigte `if`-Kaskaden.

Insbesondere die Betrachtung der tief verschachtelten Verzweigungen in `spcl_switch` und `spcl_branch` brachte einige Fehler zutage, die im Anschluss behoben werden konnten.

Hiermit wurde die Grundlage geschaffen, um auch komplexere Testprogramme als Proof-of-Concept umzusetzen. Da zu diesem Zeitpunkt noch keine vollständige Unterstützung von Schleifenstrukturen existierte, sind nun einige Testprogramme, sowohl in einer rekursiven Variante, als auch als Implementierung mit Schleifen vorhanden.

[1] Definiert als $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \text{fib}(0) = 0, \text{fib}(1) = 1$.
[2] Berechnet s und t in $\text{ggT}(a, b) = s \cdot a + t \cdot b$ mit $a, b \in \mathbb{N}$ und $s, t \in \mathbb{Z}$.

Name	Beschreibung
recn_binary_search	Binäre Suche in einem Array.
both_quicksort	Eine Implementierung von Quicksort.
loop_mersenne	Der Zufallszahlengenerator „Mersenne Twister“ [MN98].
recn_mandelbrot	Berechnet rekursiv beliebige Ausschnitte der Mandelbrot-Menge[3].
loop_mandelbrot	Die „gewöhnliche“ Variante mit Schleifen.
recn_brainfuck	Ein Brainfuck[4]-Interpreter [Rai] mit rekursiven Funktionen.
loop_brainfuck	Die entsprechende Variante mit Schleifen.

Die binäre Suche ist hierbei der einfachste und daher auch der zuerst implementierte Test. `loop_mersenne` ist in Bezug auf die Kontrollstrukturen bei etwa 60 Codezeilen noch recht einfach, die Rekonstruktion der Schleifenstrukturen ist jedoch nicht zu unterschätzen. Quicksort ist interessant, weil hierbei viele Konstrukte kombiniert werden. Verschachtelte und aufeinander folgende Schleifen, kombiniert mit Verzweigungen und Rekursion.

Mit etwa 80 Codezeilen sind die Mandelbrot- und Brainfuck-Testprogramme vergleichsweise aufwendig. `loop_mandelbrot` ist durch die stark verschachtelten Schleifen interessant, während `loop_brainfuck` für das Interpretieren eine `switch`-Anweisung nutzt, die in mehrere Verzweigungen zerfällt und darüber hinaus in einer Schleife eingebettet ist. Die Rekonstruktion dieser Kontrollstrukturen durch die Abbau-Phase ist daher recht anspruchsvoll.

7.2 Korrektheit

Zum Test der Ergebnisse wurden die mittels vFirm übersetzten Testprogramme automatisiert mit verschiedenen Eingabeparametern aufgerufen. Das Ergebnis wurde mit dem Ergebnis der Referenzprogramme verglichen, die direkt mit dem GCC-Compiler [GCC] aus demselben Quellcode erzeugt wurden.

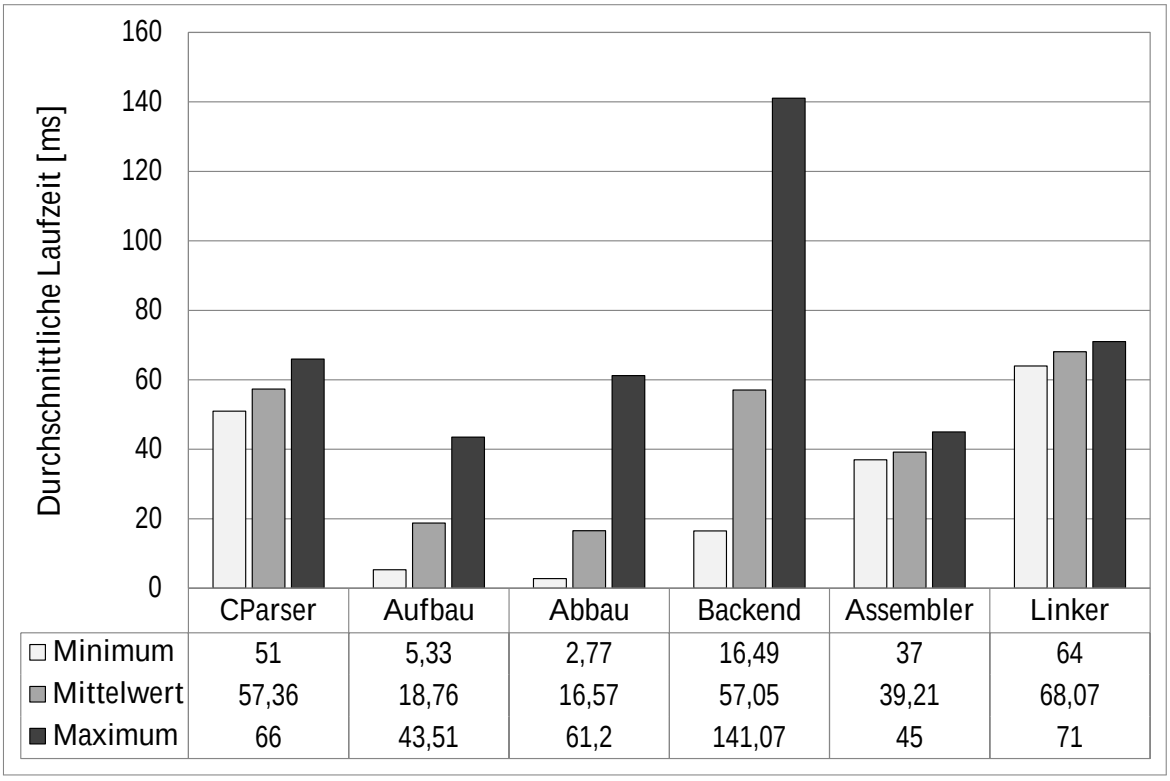
Inzwischen konnten alle genannten Testprogramme (sowie einige nicht genannte Testprogramme) mit der entwickelten vFirm-Implementierung übersetzt und erfolgreich getestet werden. Für eine umfangreichere Auswertung mit größeren Testprogrammen fehlte leider die Zeit. Diese Ergebnisse zeigen aber bereits, dass eine solide Grundlage für weitere vFirm-Entwicklungen geschaffen wurde.

7.3 Geschwindigkeit

Um die Geschwindigkeit der Transformation zu vergleichen, wurden die einzelnen Übersetzungsphasen für die oben genannten Testprogramme zeitlich erfasst. Dabei wurde zur Vermeidung von Ausreißern der Median von den jeweils 5 Durchläufen verwendet. Aus den so

[3] Oder kurz gesagt: eine Darstellung der Konvergenz von $z_{n+1} = z_n^2 + c$ auf der komplexen Zahlenebene.

[4] Eine von Urban Müller entwickelte, extrem einfache aber Turing-vollständige Programmiersprache. Sie hat das erklärte Ziel, mit einem kleinstmöglichen Compiler übersetzt werden zu können.



	CParser	Aufbau	Abbau	Backend	Assembler	Linker
Minimum	51	5,33	2,77	16,49	37	64
Mittelwert	57,36	18,76	16,57	57,05	39,21	68,07
Maximum	66	43,51	61,2	141,07	45	71

Abbildung 7.1: Laufzeit der Übersetzungsphasen

gewonnenen Werten wurde nun für jede Phase jeweils die minimale, maximale und durchschnittliche Laufzeit ermittelt. Die Ergebnisse sind in Abbildung 7.1 dargestellt[5].

Die hierbei betrachteten Phasen sind:

1. **CParser**: Compilieren des C-Programmes nach Firm.
2. **Aufbau**: Konstruktion des vFirm-Graphen aus Firm.
3. **Abbau**: Rekonstruktion des Firm-Graphen aus vFirm.
4. **Backend**: Überführen in die Assembler-Sprache durch das Firm-Backend.
5. **Assembler**: Erzeugen des entsprechenden Maschinencodes.
6. **Linker**: Einbinden von Bibliotheken. Erzeugung einer ausführbaren Datei.

Die Transformation ist also – zumindest für diese Testprogramme – immer noch vergleichsweise schnell. Weitere Verbesserungen dürften darüber hinaus noch möglich sein. Für diese erste Implementierung war die Geschwindigkeit jedoch nicht das vorrangige Ziel.

[5]Testsystem: Intel Core 2 Quad Q6600, Ubuntu Linux 10.04 auf Oracle VirtualBox 4.0.8.

7.4 Codequalität

Häufig reicht die Qualität des zurücktransformierten Programmes nicht an das Ursprungsprogramm heran. Verantwortlich dafür sind – soweit dies bisher ersichtlich ist – die folgenden zwei Punkte:

1. Ausführungsbedingungen werden in vFirm weitaus strenger gehandhabt, als in Firm. Während Firm Knoten teilweise spekulativ ausführt, erzeugt die Transformation via vFirm unter Umständen auch für einzelne Knoten zusätzliche Verzweigungen.
2. Die Kombination von `loop`-Knoten bedarf weiterer Verbesserungen. Häufig werden mehrere `loop`-Knoten derselben Ursprungs-Schleife bei der Rücktransformation nicht kombiniert. Dies liegt meist daran, dass der Spielraum beim Platzieren der Knoten nicht ausgenutzt wird, um diese derselben Region zuzuordnen.

Dies sorgt für eine Fragmentierung der Schleifen, welche sich negativ auf das Ergebnis auswirkt. Im Extremfall zerfällt dabei eine Schleife in mehrere nahezu äquivalente Schleifen, die sich nur durch wenige Knoten unterscheiden.

Eine lokal beschränkte Suche nach vergleichbaren `loop`-Knoten, die vor der eigentlichen Platzierung stattfindet, könnte bereits eine starke Verbesserung zur Folge haben und wäre für eine sinnvolle Bewertung der Ergebnisse eigentlich erforderlich. Die Schwierigkeit liegt hierbei in der Tatsache, dass das Verschieben eines `loop`-Knotens auch alle Abhängigkeiten verschieben kann und die Auswirkungen nur schwer überschaubar sind. Aus Zeitgründen konnte noch kein entsprechender Algorithmus entworfen und implementiert werden.

Für die Schleifen spielt aber auch der erste genannte Punkt eine Rolle. Könnte man die Bedingungen für einen `loop`-Knoten lockern, dann erhielte man auch einen größeren Spielraum bei dessen Platzierung. Wenn zwei prinzipiell kombinierbare `loop`-Knoten dieselbe Vater-Region haben (was durchaus denkbar ist), dann wäre das Lockern der Bedingungen alleine schon ausreichend.

Duplikation und Zustandswerte Außerhalb von Schleifen ist die Duplikation von Knoten mit Nebeneffekten unproblematisch. Dies liegt daran, dass Knoten stets nur eine Zustands-Abhängigkeit besitzen können und sich der „Auswertungspfad“ für Zustände somit nur an γ-Knoten verzweigen kann. Da diese höchstens eine ihrer Abhängigkeiten auswerten, kommt es auch bei Duplikaten nur zur Auswertung einer der Kopien. Nebeneffekte können damit nicht mehrfach auftreten.

Problematischer sind in diesem Zusammenhang Schleifen, die nicht mehr zusammengefasst werden können. Bei den Testprogrammen ist diese Situation zwar nicht aufgetreten, aber

theoretisch kann es passieren, dass verschiedene nicht-zusammengefasste Schleifen auf denselben zustandsbehafteten Knoten zugreifen und so eine Duplikation erzwingen. Voraussichtlich entsteht hierbei ein ungültiger Graph, da nur der Zustand einer Kopie weiterverwendet wird. Die korrekte Zusammenfassung von Schleifen ist damit auch für die Korrektheit des Programmes erforderlich.

Der technische Bericht zur PEG-Darstellung [TSTL11] geht auch noch einmal auf das Thema ein und es ist anzunehmen, dass sich das Problem durch eine entsprechende Optimierung der Schleifen beheben lässt. Es wäre ansonsten auch noch zu überlegen, ob man Informationen über die Ursprungsschleife an den η-Knoten hinterlegt, um später zusammengehörige `loop`-Knoten zu identifizieren und so die Optimierung zu erleichtern.

8 Fazit

8.1 Zusammenfassung

Die neue Zwischendarstellung vFirm schafft durch die Reduktion auf die reinen Datenabhängigkeiten zweifellos einen größeren Spielraum, den der Compiler für Optimierungen nutzen kann. Gleichzeitig steigen mit dem gewonnenen Spielraum aber auch die Anforderungen an eine Rücktransformation oder ein hypothetisches Backend, dass direkt mit der vFirm-Darstellung arbeitet.

Entsprechend war die Implementierung der Rücktransformation im Rahmen dieser Arbeit keine leichte Aufgabe und hat bis zu diesem Stand auch schon einige Umstrukturierungen hinter sich. Mit Sicherheit werden auch noch weitere Entwicklungen notwendig sein, um das volle Potential der vFirm-Darstellung ausschöpfen zu können, die Grundlage hierfür ist nun zumindest vorhanden.

8.2 Weitere Schritte

Dieser Abschitt soll einige Anregungen geben, welche weiteren Schritte zu einer Verbesserung der Implementierung sinnvoll sein könnten.

Irreduzibler Steuerfluss Momentan ist es nicht möglich, Schleifen mit mehr als einem Schleifenkopf nach vFirm zu transformieren. Wie allerdings in Abschnitt 5.3.6 erwähnt, lässt sich dies durch eine Vorverarbeitung des Graphen ermöglichen. Die Umsetzung hiervon dürfte relativ unproblematisch sein, allerdings tritt diese Situation auch eher selten auf.

Schleifen besser kombinieren Wie in Abschnitt 7.4 schon angemerkt, leidet die Codequalität im Moment *primär* unter der Tatsache, dass `loop`-Knoten zu selten zu einer Schleife zusammengefasst werden. Die Platzierung von `loop`-Knoten sollte die Möglichkeiten zur Kombination von Schleifen berücksichtigen, um einer Fragmentierung der Schleifen entgegenzuwirken. Das Lockern der entsprechenden Gating-Bedingungen sollte ebenfalls in Betracht gezogen werden.

Duplikationen vermeiden Interessant wäre es auch, die Duplikation von Knoten unter gewissen Umständen zu vermeiden. Bei den Schleifen lässt sich dies nur durch eine bessere Kombination von Schleifen erreichen, aber unter Umständen ist es möglich, mehrere Regions-Hinweise zu kombinieren, die im Moment noch eine Duplikation der Knoten zur Folge haben. Ein einfaches Beispiel für eine solche Situation ist in Abbildung 21 dargestellt.

Listing 21 Vermeidbare Duplikation

```
1: var a
2: if P then
3:     a = x + 1
4:     y = a * 2
5: end if
6: ...
7: if P then
8:     z = a * 3
9: end if
```

Dabei wird die Addition für den Wert a in beiden Verzweigungen benötigt. Können diese nicht zusammengefasst werden, dann wird für beide Regionen ein Hinweis erzeugt. Da die Addition aber die Bedingung P voraussetzt, kann sie nicht in die übergeordnete Region verschoben werden, so dass die Hinweise nicht kombiniert werden können. Will man also aus der zweiten Verzweigung auf a zugreifen, so müsste man die Addition entweder spekulativ ausführen oder hinter der ersten Verzweigung einen ϕ-Knoten konstruieren, der einen Zugriff ermöglicht. Allerdings ist im Fall $\neg P$ kein Wert für a definiert.

Dies ist zur Zeit experimentell implementiert, bedarf jedoch noch weiterer Tests. Für die verwendeten Testprogramme hielt sich der Effekt allerdings in Grenzen.

Gating-Bedingungen vereinfachen Die bisher verwendete Methode zur Vereinfachung der Gating-Bedingungen ist relativ einfach. Es ist daher möglich, dass eine verbesserte Strategie Verzweigungen vermeiden kann. Dies dürfte primär dann eine Verbesserung bewirken, wenn man tief verzweigte Programme betrachtet.

Gating-Bedingungen lockern Im Moment ist der Abbau darauf ausgelegt, Ausführungsbedingungen exakt zu befolgen. Es kommt mitunter vor, dass eigene Verzweigungen für einzelne Knoten geschaffen werden, obwohl eine spekulative Auswertung des Knotens viel günstiger wäre. Es ist daher zu überlegen, ob man die spekulative Auswertung von Knoten bei der Berechnung der Gating-Bedingungen berücksichtigen sollte.

Optimierungen auf vFirm Durch die vFirm-Darstellung bieten sich ganz neue Möglichkeiten der Programmoptimierung, insbesondere da man nicht mehr an starre Steuerflussstrukturen gebunden ist und auch schleifenübergreifende Optimierungen einfacher ausdrücken kann.

Es ist daher interessant, Optimierungen auf der vFirm-Darstellung selbst durchzuführen und mit den klassischen Ansätzen zu vergleichen. Aktuelle und zukünftige Entwicklungen bei den VSDG- und PEG-Darstellungen könnten hier auch interessante Ansätze liefern.

Literaturverzeichnis

[AU77] Alfred V. Aho und Jeffrey D. Ullman. *Principles of Compiler Design*. Boston, MA, USA: Addison-Wesley, 1977. ISBN: 978-0201000221.

[BBZ11] Matthias Braun, Sebastian Buchwald und Andreas Zwinkau. *Firm—A Graph-Based Intermediate Representation*. 2011. URL: http://pp.info.uni-karlsruhe.de/uploads/publikationen/braun11wir.pdf.

[BMO90] Robert A. Ballance, Arthur B. Maccabe und Karl J. Ottenstein. „The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages“. In: *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM, 1990, 257–271.

[CFRWZ91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman und F. K Zadeck. „Efficiently computing static single assignment form and the control dependence graph“. In: *ACM Transactions on Programming Languages Systems* 13.4 (Okt. 1991), 451–490.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein und Joe D. Warren. „The Program Dependence Graph and Its Use in Optimization“. In: *ACM Transactions on Programming Languages Systems* 9.3 (Juli 1987), 319–349.

[GCC] GCC Team. *GCC: the GNU Compiler Collection*. URL: http://gcc.gnu.org.

[HU74] M. S. Hecht und J. D. Ullman. „Characterizations of Reducible Flow Graphs“. In: *Journal of the ACM* 21.3 (Juli 1974), 367–375.

[IPD] IPD Karlsruhe, Lehrstuhl Programmierparadigmen. *libFirm Projekt-Webseite*. URL: http://www.libfirm.org/.

[JM03] Neil Johnson und Alan Mycroft. „Combined code motion and register allocation using the value state dependence graph“. In: *Proceedings of the 12th International Conference on Compiler Construction*. Springer Verlag, 2003, 1–16.

[Law07] Alan C. Lawrence. *Optimizing compilation with the Value State Dependence Graph*. Techn. Ber. UCAM-CL-TR-705. University of Cambridge, Computer Laboratory, Dez. 2007. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-705.html.

[Lin02] Götz Lindenmaier. *libFirm: A Library for Compiler Optimization Research Implementing Firm*. Techn. Ber. 2002-5. Universität Karlsruhe, Sep. 2002. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/5002002.

[MN98] M. Matsumoto und T. Nishimura. „Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator“. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (Jan. 1998), 3–30.

[Rai] Brian Raiter. *Brainfuck: An Eight-Instruction Turing-Complete Programming Language*. URL: `http://www.muppetlabs.com/~breadbox/bf`.

[Ram02] G. Ramalingam. „On loops, dominators, and dominance frontiers“. In: *ACM Transactions on Programming Languages Systems* 24.5 (Aug. 2002), 455–490.

[RWZ88] Barry K. Rosen, Mark N. Wegman und F. Kenneth Zadeck. „Global Value Numbers and Redundant Computations“. In: *"POPL 1988"*. 1988, S. 12–27.

[TLB99] Martin Trapp, Götz Lindenmaier und Boris Boesler. *Documentation of the Intermediate Representation Firm*. Techn. Ber. 1999-44. Universität Karlsruhe, Dez. 1999. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/302599`.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock und Sorin Lerner. „Equality saturation: a new Approach to Optimization“. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2009, 264–276. URL: `http://cseweb.ucsd.edu/~rtate/publications/eqsat/`.

[TSTL11] Ross Tate, Michael Stepp, Zachary Tatlock und Sorin Lerner. *Translating between PEGs and CFGs*. Techn. Ber. San Diego: University of California, 2011. URL: `http://cseweb.ucsd.edu/~rtate/publications/eqsat/`.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst und Bjarne Steensgaard. „Value dependence graphs: Representation without taxation“. In: *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1994, 297–310.