Entwurf und Implementierung eines Generators für Datenflußanalysatoren

Florian Martin

Diplomarbeit nach einem Thema von
Prof. Dr. Reinhard Wilhelm

Fachbereich 14
Universität des Saarlandes, Saarbrücken

Juni 1995

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit eigenständig und nur mit Hilfe der angegebenen Quellen angefertigt habe.
Die Diplomarbeit wurde noch keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Saarbrücken, den 14. Juni 1995

An dieser Stelle möchte ich Herrn Prof. R. Wilhelm für das interessante Thema und Martin Alt für die Betreuung meiner Diplomarbeit und die vielen wertvollen Anregungen in langen Gesprächen danken.
Mein Dank gilt ebenso meiner Freundin Verena und meinen Eltern für ihre Unterstützung während meines ganzen Studiums.
Für das exakte Korrekturlesen meiner Arbeit bedanke ich mich bei Christian Probst und Christian Fecht sowie Gerd und Heide Horn.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Optimierung von Programmen ist ein wichtiges Teilgebiet des modernen Übersetzerbaus. Deshalb wurden viele Anstrengungen unternommen, um Optimierungsalgorithmen zu entwickeln und zu implementieren und ihre Korrektheit zu beweisen. Es gibt globale und lokale Optimierungen: Die lokalen Verfahren betrachten immer nur einen Teil des Programms und ersetzen diesen durch einen effizienteren. Globale Algorithmen hingegen benötigen auf einmal Informationen über das gesamte Programm. Daher ist ihre Implementierung besonders schwierig und zeitaufwendig.

Wenn das Interesse an Analysen wächst, so hängt das nicht nur mit den gestiegenen Ansprüchen an die Effizienz von erzeugtem Code zusammen, sondern auch mit der Hardwareentwicklung der letzten Jahre. Die Erhöhung der Geschwindigkeit moderner Prozessoren stößt immer mehr an die Grenzen des physikalisch Möglichen. Daher wird seit Jahren die Parallelverarbeitung vorangetrieben, sei es unter Verwendung von mehrstufigen Pipelines, ohne die heute kein Prozessor mehr auskommt, oder seien es massiv parallele Rechner. Die Erfahrung zeigt, daß es den Benutzern nicht zuzumuten ist, Algorithmen diesen Architekturen anzupassen. Deshalb muß die Anpassung weitestgehend automatisch erfolgen, was eine gute Analysephase unabdingbar macht.

Globale Analysatoren sind verfahrensbedingt sehr komplex, da sie Informationen über die Laufzeit von Programmen berechnen sollen, wobei die Terminierung der Analyse – im Gegensatz zur Ausführung – garantiert sein muß. Daher ist ihre Implementierung sehr zeitraubend und kompliziert und damit fehleranfällig. Auch ist der entstehende Code meist nur sehr schwer zu warten, da er zur Steigerung der Performance in einem nicht-modularen Stil implementiert wird. Daher ist es wünschenswert, auch in diesem Teilgebiet des Übersetzerbaus Werkzeuge einzusetzen, wie sie bereits für fast jede andere Phase des Compilers zur Verfügung stehen. So benutzt man für die Generierung der lexikalischen und syntaktischen Analysen seit Jahrzehnten Generatoren wie `lex` und `yacc`, und auch die automatische Erzeugung von Codegeneratoren gewinnt in den letzten Jahren immer mehr an Bedeutung. Nur auf dem Gebiet der Erzeugung von globalen Analysatoren und Transformatoren zur anschließenden Optimierung gibt es bisher kaum brauchbare Werkzeuge.

Deshalb wurde im Rahmen dieser Diplomarbeit ein Werkzeug namens PAG entwickelt, das eine leistungsfähige Möglichkeit bietet, um den Übersetzerbauer von der fehleranfälligen Routinearbeit zu entlasten.

1.1 PAG

PAG ist ein **P**rogramm **A**nalysator **G**enerator, der es erlaubt, Analysealgorithmen auf hohem abstrakten Niveau zu spezifizieren und daraus die benötigten Analysatoren automatisch zu generieren. Diese dienen nicht nur dazu, eine Analyse vor ihrer endgültigen Implementierung zu testen, vielmehr sind die erzeugten Analysatoren so effizient, daß sie direkt in einem Compiler verwendet werden können.

Das verringert nicht nur den Zeitaufwand, um eine Analyse zu implementieren, sondern erleichtert dem Programmierer die Wartung und Fehlersuche in der Analyse. Außerdem bietet PAG die Möglichkeit, wesentliche Parameter des Analysators in kürzester Zeit zu verändern, somit Analysen zu beschleunigen und unter verschiedenen Lösungsverfahren das beste auszuwählen.

1.2 Das Projekt COMPARE

Diese Arbeit entstand im Rahmen des ESPRIT Projekts Nr. 5399 COMPARE. Das Compilermodell des Projektes CoSy (COmpiler SYstem) wurde mit dem Ziel entworfen, eine flexible Anordnung von Phasen (den sogenannten Engines) in einem Compiler mit kurzen Generierungs– und Übersetzungszeiten zu ermöglichen. Dies wird dadurch erreicht, daß die grundlegenden Phasen, nämlich die vom Benutzer implementierten oder von anderen Werkzeugen (wie z.B. PAG) generierten Algorithmen, Schritt für Schritt zu immer komplexeren Phasen zusammengefaßt werden.

Der Nutzen des CoSy-Modells besteht darin, daß es dem Benutzer möglich wird, eine Phase des Compilers zu implementieren und diese ohne Änderung des Quellcodes in verschiedenen Compilerkontexten zu benutzen. Das können z.B. verschiedene andere Engines oder auch eine parallele Umgebung sein. Da PAG in der Lage ist, Code zu erzeugen, der die Speicherzugriffsfunktionen von CoSy aus dem DMCP (Data Manipulation and Control Package) verwendet, ist es möglich, die generierten Analysatoren in verschiedenen Compilern wie z.B. ANSI C, Fortan 90 oder Modula

Compilern einzusetzen, ohne sie jedesmal aufs neue anpassen zu müssen.
Eine genaue Beschreibung des CoSy Compilermodells ist in [AAvS94] zu finden.

1.3 Gliederung der Arbeit

Kapitel 2 soll als Basis für die folgenden theoretischen Kapitel einen kurzen Überblick über die wichtigsten Techniken der globalen Analyse geben. In Kapitel 3 werden dann die mathematischen Grundlagen der Analyse behandelt. Anschließend wird in Kapitel 4 die klassische Datenflußanalyse mit möglichen Lösungsalgorithmen genau untersucht. In Kapitel 5 wird die unerläßliche Erweiterung des klassischen Rahmens auf Programme mit Prozeduren vorgenommen.

In den folgenden Kapiteln wird die Arbeit sich mit der Generierung der vorher dargelegten Lösungswege beschäftigen: In Kapitel 6 werden kurz die allgemeinen Grundlagen dargestellt, Kapitel 7 beschreibt die Spezifikationssprachen von PAG. In Kapitel 8 wird auf einige Implementierungsdetails eingegangen, sodann werden in Kapitel 9 praktische Ergebnisse von PAG generierten Analysatoren dargestellt.

Anschließend werden in Kapitel 10 vergleichbare Arbeiten vorgestellt sowie ein Ausblick in die Zukunft von PAG gegeben. In Anhang A ist die vollständige Spezifikation eines Analysators angegeben, und in Anhang B werden einige Meßwerte aufgelistet.

Kapitel 2

Konzepte der globalen Optimierung

2.1 Ziele

In diesem Kapitel soll ein Überblick über die Hauptprobleme, Konzepte und Techniken der globalen Analyse gegeben werden. Es wird also diejenige Phase eines Compilers betrachtet, in der Informationen über mögliche Verhalten des Programms zur Laufzeit berechnet werden. Diese Informationen werden benötigt, um verschiedene semantikerhaltende Transformationen des Programms durchführen zu können. Es kann sich dabei um Optimierungen handeln, wie das Entfernen redundanter Berechnungen oder die Parallelisierung für einen Supercomputer. Die hier vorgestellten Techniken sind bei allen Arten von kontrollbasierten Verfahren auch außerhalb von Compilern einsetzbar, wie z.B. bei der automatischen Verifikation von Software.

Da viele der gesuchten Informationen nicht berechenbar sind, gilt es, eine Annährung an das optimale Ergebnis zu finden. Diese sollte, wenn sie schon nicht zu 100% richtig ist, zumindest so korrekt sein, daß damit die darauf basierenden Transformationen semantikerhaltend sind. Solche Informationen, die zwar korrekt aber nicht unbedingt optimal sind, nennt man **konservativ**. So ist die Aussage, „Anweisung s_1 und s_2 können nicht gleichzeitig ausgeführt werden", für einen Parallelisierer auch dann konservativ, wenn sie falsch ist, da die Anweisungsfolge $s_1; s_2$ in diesem Fall nicht transformiert wird.

```
int fac(int x) {
   int z;

   z = 1;
   while(x > 1) {
      z = z*x;
      x = x-1;
   }
   return(z);
}
```

Abbildung 2.1: Die nicht rekursive Fakultätsfunktion

2.2 Algorithmen

2.2.1 Datenflußanalyse

Die am weitesten verbreitete Technik, um statisch Laufzeitinformationen über ein Programm zu erhalten, ist die sogenannte **Datenflußanalyse**. Für ein gegebenes Programm wird zunächst ein Graph berechnet, der alle zur Laufzeit möglichen Pfade durch ein Programm repräsentiert. Hier treten bereits erste Ungenauigkeiten auf, da nicht alle in einem solchen Graph beschriebenen Pfade auch tatsächlich mögliche Ausführungspfade sind. Aus technischen Gründen, auf die später näher eingegangen wird, wird der Graph um eindeutige Start– und Endknoten erweitert. Als Beispiel ist in Abbildung 2.2 der **Kontrollflußgraph** des Programms aus Bild 2.1 dargestellt.

Das weitere Vorgehen soll nun am Beispiel „**verfügbarer Definitionen**“ gezeigt werden. Eine Definition einer Variablen x eines Programms ist eine Anweisung, die x einen Wert zuweist. Wegen der konservativen Abschätzung werden auch solche Anweisungen dazugezählt, die x eventuell einen Wert zuweisen, wie z.B. ein Prozeduraufruf, in dem x als Referenzparameter übergeben wird. Man sagt, eine Definition d ist an einem Programmpunkt[1] p verfügbar, wenn es einen Pfad im Kontrollflußgraphen von dem direkt auf d folgenden Punkt nach p gibt, wobei die Definition d auf diesem Pfad nicht durch eine erneute Definition derselben Variablen „zerstört“ wird.

Zur Berechnung beschriftet man Kanten im Kontrollflußgraphen mit (Meta-) Variablen, welche später die zu berechnende Datenflußinformation (in diesem Fall Mengen von Definitionen) enthalten werden. Die Knoten werden mit Funktionen versehen, die aus allen eingehenden Werten einen neuen Wert berechnen. Defini-

[1] Ein Programmpunkt ist eine Stelle in einem Programm, direkt vor bzw. hinter einer Anweisung.

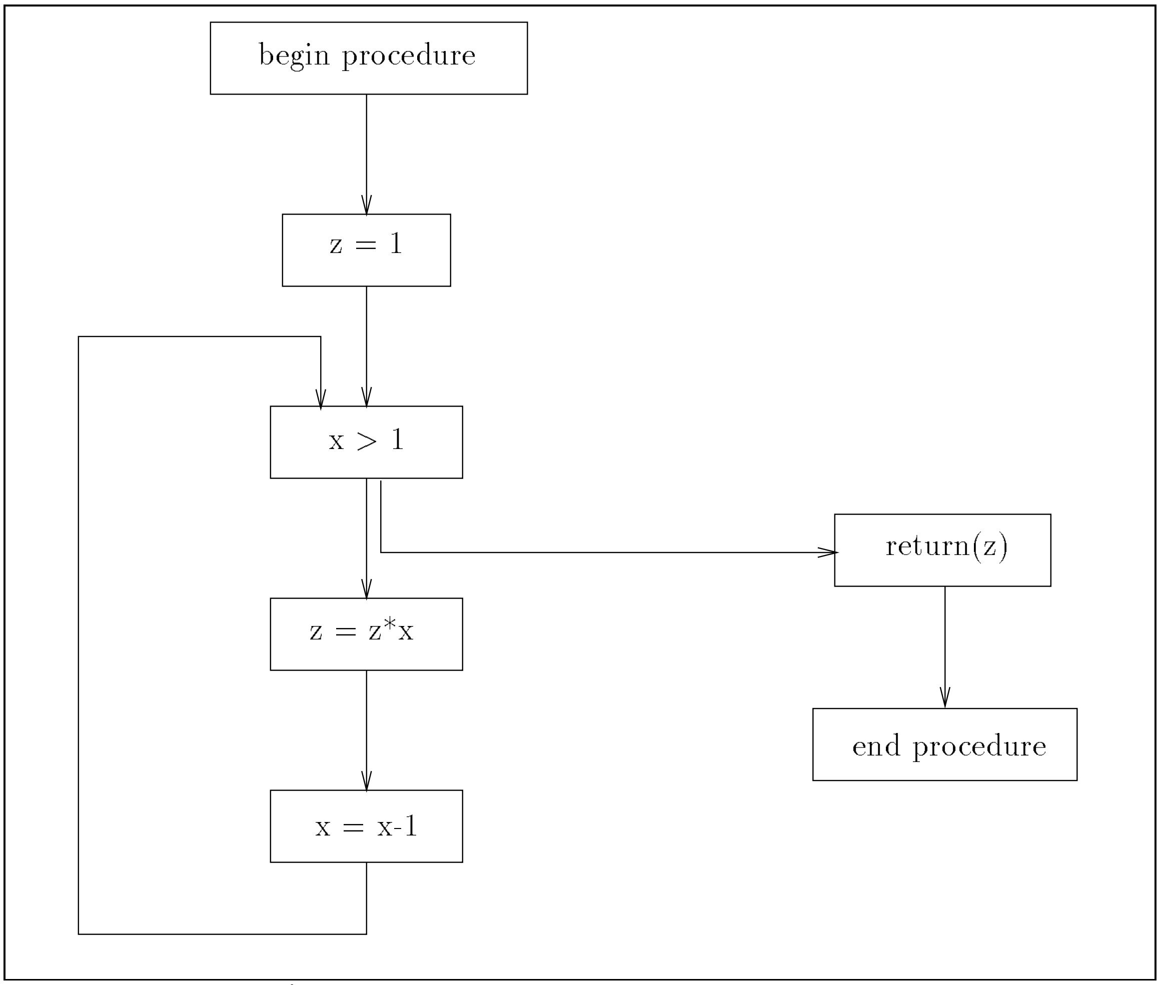

Abbildung 2.2: Der Kontrollflußgraph zu Bild 2.1

tionen werden mit der Nummer des zugehörigen Knotens bezeichnet. Da an einem Knoten alle Definitionen sichtbar sind, die auf verschiedenen Pfaden zu ihm gelangen, muß die Vereinigung über alle eingehenden Informationen gebildet werden. Falls der Knoten selbst eine Definition d enthält, müssen hieraus alle Definitionen entfernt werden, die von d zerstört werden, und anschließend muß d selbst hinzugefügt werden. Dies führt zu dem in Bild 2.3 gezeigten beschrifteten Graphen, auch **Datenflußgraphen** genannt.

Dieser Graph stellt nun ein rekursives Gleichungssystem dar, das mit mathematischen Methoden gelöst werden kann. Die sich so ergebende Lösung ist in Tabelle 2.1 dargestellt.

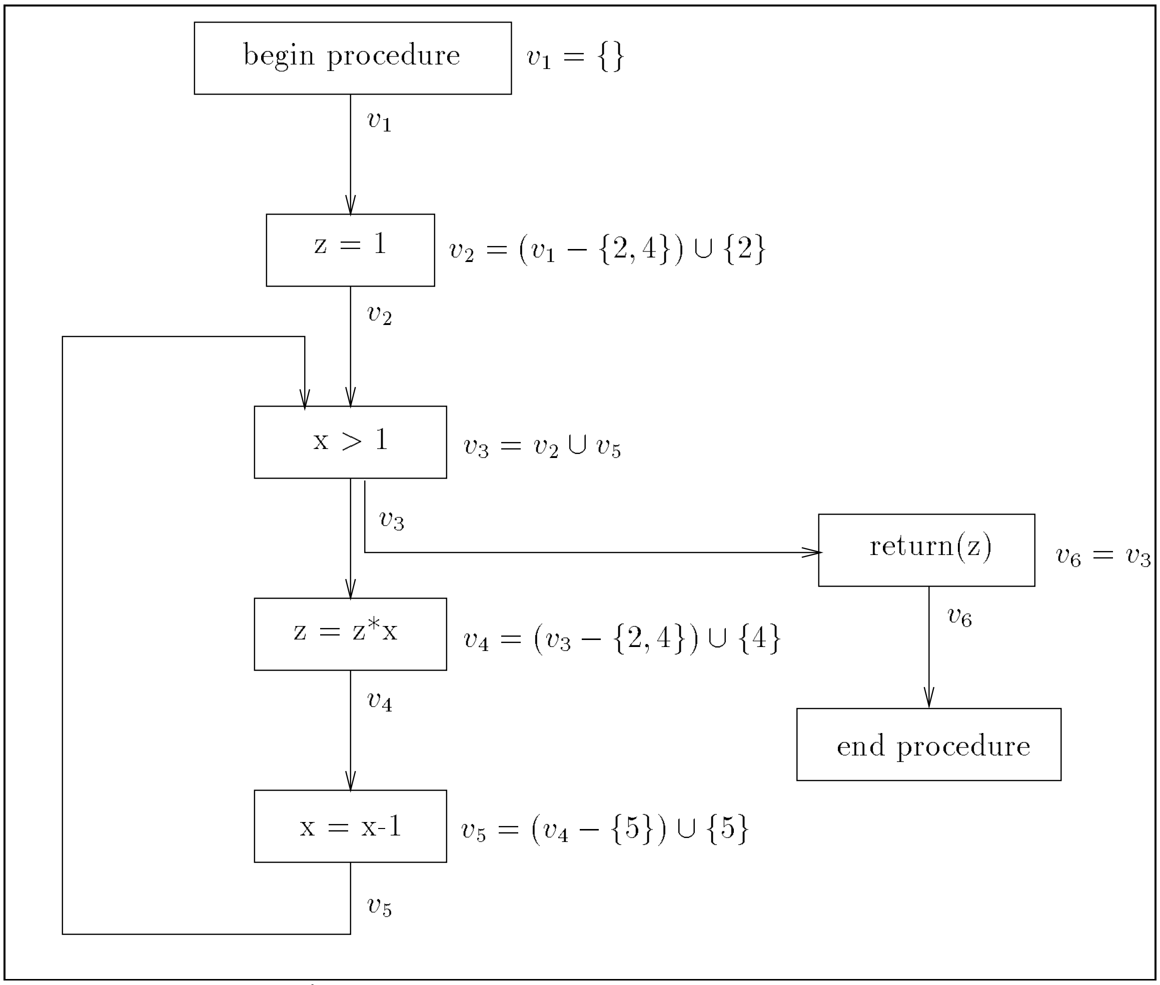

Abbildung 2.3: Der Datenflußgraph zu Bild 2.1

Variable	Wert
v_1	$\{\}$
v_2	$\{2\}$
v_3	$\{2,4,5\}$
v_4	$\{4,5\}$
v_5	$\{4,5\}$
v_6	$\{2,4\}$

Tabelle 2.1: Die Ergebnisse des rekursiven Gleichungssystems aus Bild 2.3

2.2.2 Abstrakte Interpretation

In der abstrakten Interpretation versucht man eine **konkrete Semantik**, die auf konkreten Werten basiert, durch eine **abstrakte Semantik**, die auf abstrakten Werten arbeitet, zu ersetzen. Die beiden Wertemengen sollen eine feste Beziehung besitzen: Für jeden konkreten Wert d soll eine Abstraktion $\overline{d}$ existieren, die d beschreibt. Man sagt, $\overline{d}$ abstrahiert d und drückt dies durch eine **Abstraktionsrelation** δ aus: $d\ \delta\ \overline{d}$. Jede Operation $\circ$ der konkreten Semantik muß eine Entsprechung $\overline{\circ}$ in der abstrakten Semantik besitzen. Für diese muß gelten:

$$\text{Falls } d_1 \circ d_2 = d \text{ und } d_1\,\delta\,\overline{d_1} \text{ und } d_2\,\delta\,\overline{d_2} \text{ folgt } d\,\delta\,\overline{d_1}\,\overline{\circ}\,\overline{d_2}$$

Dann ist gewährleitet, daß $\overline{\circ}$ die Operation $\circ$ korrekt abstrahiert.

Diese abstrakte Semantik versucht man so zu wählen, daß ihre Berechnungen immer terminieren und die Abstraktion gute Rückschlüsse auf das Verhalten des Programms zuläßt.

Beispiel 2.2.1
Hier soll die Standardsemantik mit den Operationen ‚+' und ‚*' für Ausdrücke durch eine Semantik ersetzt werden, die versucht, die Vorzeichen der Ausdrücke zu berechnen. Um Werte größer, kleiner oder gleich null zu beschreiben, wird der abstrakte Wertebereich als $\{>, <, 0, ?\}$ gewählt. Das Fragezeichen steht für Werte unbekannten Vorzeichens. Die Operationen $\overline{+}$ und $\overline{*}$ sind dann durch folgende Tabellen gegeben:

$\overline{+}$	$>$	$<$	0	?
$>$	$>$	?	$>$	?
$<$	?	$<$	$<$	?
0	$>$	$<$	0	?
?	?	?	?	?

$\overline{*}$	$>$	$<$	0	?
$>$	$>$	$<$	0	?
$<$	$<$	$>$	0	?
0	0	0	0	0
?	?	?	0	?

Man kann also für einen Ausdruck von den konkreten Werten abstrahieren, die abstrakte Semantik anwenden und von deren Ergebnissen auf eine Menge von konkreten Werten schließen, die als Ergebnisse möglich sind. □

Kapitel 3

Grundlagen

3.1 Verbandstheorie

3.1.1 VPOs, Verbände und stetige Funktionen

Da die vorgestellten Analysemethoden auch bei Schleifen in Programmen anwendbar sein sollen, wird eine klassische Methode verwendet, die unter bestimmten Voraussetzungen garantiert, daß der Algorithmus terminiert: die Fixpunktiteration. Ihre mathematischen Grundlagen werden im folgenden genauer erläutert.

Definition 3.1.1 (partielle Ordnung, totale Ordnung)
Sei P eine Menge. Eine binäre Relation $\sqsubseteq\ \subseteq P \times P$ auf P heißt *partielle Ordnung* auf P, wenn sie folgende Eigenschaften besitzt:

1. Reflexivität: $\forall p \in P : p \sqsubseteq p$
2. Transitivität: $\forall p, q, r \in P : p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$
3. Antisymmetrie: $\forall p, q \in P : p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$.

Sie heißt *totale Ordnung*, wenn sie zusätzlich $\forall p, q \in P : p \sqsubseteq q \vee q \sqsubseteq p$ erfüllt. Mit $\sqsubset$ wird die reflexive Reduktion von $\sqsubseteq$ bezeichnet. Die Menge P, versehen mit der partiellen Ordnung $\sqsubseteq$, heißt *partiell geordnete Menge*, in Zeichen $(P, \sqsubseteq)$. $\square$

Eine partielle Ordnung $\sqsubseteq$ auf einer Menge P kann durch einen gerichteten zyklenfreien Graphen $G = (P, E)$ dargestellt werden, für den gilt: $p_1 \sqsubseteq p_2 \Leftrightarrow$ es gibt einen Pfad von p_2 nach p_1 in G. Beispielsweise ist in Abbildung 3.1 die Relation „Nachfahre von“ dargestellt.

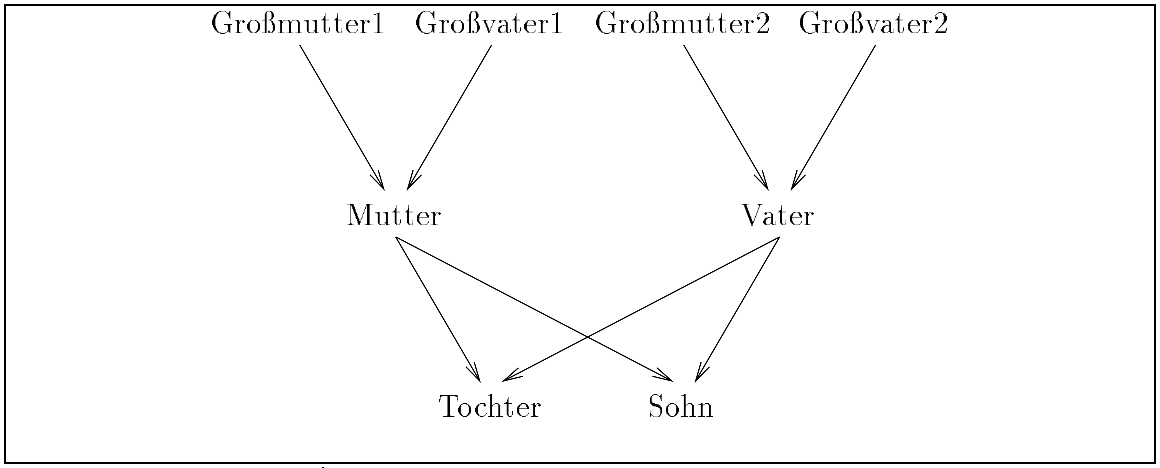

Abbildung 3.1: Die Relation „Nachfahre von“

Definition 3.1.2 (Obere Schranke)
Sei $(P, \sqsubseteq)$ eine partiell geordnete Menge und $X \subseteq P$. $p \in P$ heißt eine *obere Schranke* von X, wenn $\forall q \in X : q \sqsubseteq p$ ist.
p heißt *kleinste obere Schranke* von X (in Zeichen $\bigsqcup X$) wenn

1. p obere Schranke von X ist, und

2. für alle oberen Schranken q von X $p \sqsubseteq q$ ist.

Falls X nur zwei Elemente x, y enthält, wird statt $\bigsqcup X$ auch $x \sqcup y$ geschrieben. $\bigsqcup$ heißt *Vereinigung* oder *join*.

Untere Schranken werden analog definiert. $\bigsqcap$ heißt *Schnitt* oder *meet*. □

Ist $\sqsubseteq$ eine partielle Ordnung, dann sind die kleinste obere und die größte untere Schranke eindeutig bestimmt, sofern sie existieren.

Ist eine partielle Ordnung durch einen gerichteten zyklenfreien Graphen gegeben, so kann man die kleinste obere Schranke und die größte untere Schranke einer Menge A durch graphische Begriffe charakterisieren: $\bigsqcup A$ ist der nächste gemeinsame Vorfahre der Elemente aus A, $\bigsqcap A$ der nächste gemeinsame Nachfahre.

Definition 3.1.3 (ω-Kette, vollständige partielle Ordnung)
Sei $(P, \sqsubseteq)$ eine partiell geordnete Menge. Eine *ω-Kette*[1] einer partiellen Ordnung ist eine aufsteigende Kette aus Elementen von P der Form $p_0 \sqsubseteq p_1 \sqsubseteq \cdots \sqsubseteq p_i \sqsubseteq \cdots$. Eine streng aufsteigende ω-Kette hat die Form $p_0 \sqsubset p_1 \sqsubset \cdots \sqsubset p_i \sqsubset \cdots$.

[1] Mit ω bezeichnet man die (total) geordnete Menge $(\mathbb{N}, \leq)$

$(P, \sqsubseteq)$ heißt *vollständige partielle Ordnung* (abgekürzt VPO), wenn für jede ω-Kette $p_0 \sqsubseteq p_1 \sqsubseteq \cdots \sqsubseteq p_i \sqsubseteq \cdots$ die kleinste obere Schranke der Menge $\{p_n | n \in \omega\}$ in P liegt. Diese wird auch geschrieben als $\bigsqcup_{n\in\omega} p_n$.
Wir nennen $(P, \sqsubseteq)$ *VPO mit Null*, wenn P ein kleinstes Element $\bot$, genannt Null besitzt. □

Die partielle Ordnung aus Abbildung 3.1 ist vollständig. Dies gilt ebenso für jede andere endliche partiell geordnete Menge, da die kleinste obere Schranke einer endlichen Kette immer ihr letztes Element ist.
Die Ordnung $\leq$ auf den natürlichen Zahlen $\mathbb{N}$ ist nicht vollständig, da die Kette $1 \leq 2 \leq 3 \leq \cdots$ keine obere Schranke in $\mathbb{N}$ besitzt.

Definition 3.1.4 (Aufsteigende Kettenbedingung)
Man sagt, eine vollständig partiell geordnete Menge $(P, \sqsubseteq)$ erfüllt die *aufsteigende Kettenbedingung*, wenn jede streng aufsteigende ω-Kette endlich ist. □

Definition 3.1.5 (Vollständiger Verband)
Eine partiell geordnete Menge $(P, \sqsubseteq)$ heißt *vollständiger Verband*, wenn jede Teilmenge von P eine kleinste obere und eine größte untere Schranke besitzt; Verbände werden auch als Sechstupel $(P, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ geschrieben, wobei $\bot = \bigsqcap P$ das kleinste und $\top = \bigsqcup P$ das größte Element ist. □

Die Operationen $\sqcup$ und $\sqcap$ eines vollständigen Verbandes $(P, \bot\top, \sqsubseteq, \sqcup, \sqcap)$ haben folgende Eigenschaften $(x, y, z \in P)$:

1. Idempotenz: $x \sqcap x = x$, $x \sqcup x = x$
2. Kommutativität: $x \sqcap y = y \sqcap x$, $x \sqcup y = y \sqcup x$
3. Assoziativität: $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$, $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$
4. Absorption: $x \sqcup (x \sqcap y) = x$, $x \sqcap (x \sqcup y) = x$
5. $x \sqcup y = y \Leftrightarrow x \sqcap y = x \Leftrightarrow x \sqsubseteq y$

Aus der Definition ist zu sehen, daß jeder vollständige Verband eine vollständig partiell geordnete Menge ist: Jede Kette ist insbesondere eine Teilmenge und hat somit eine kleinste obere Schranke.
Man kann jede Menge $M = \{m_1, m_2, m_3, \cdots\}$ partiell ordnen, indem man die Gleichheitsrelation als partielle Ordnung benutzt. Diese Ordnung wird auch als diskrete oder flache partielle Ordnung bezeichnet, da alle Elemente beziehungslos nebeneinander liegen (siehe Abbildung 3.2).

Abbildung 3.2: Graph der flachen partiellen Ordnung auf der Menge M

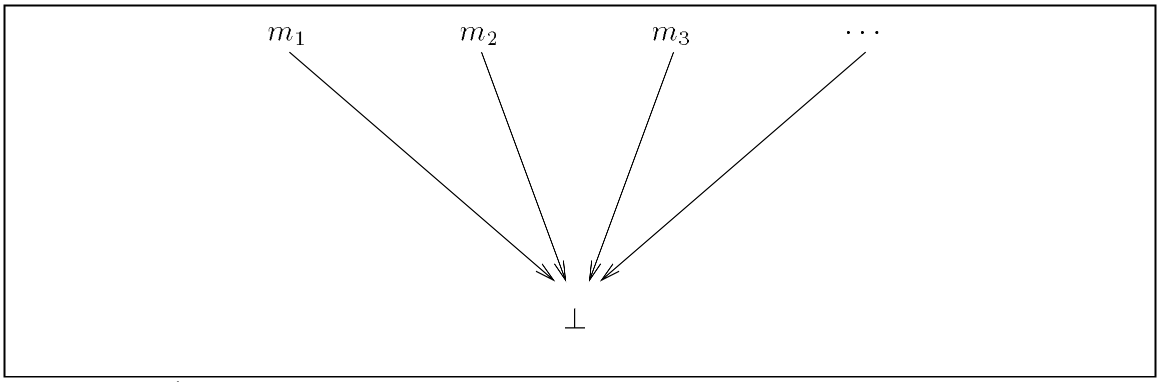

Abbildung 3.3: Graph der flachen partiellen Ordnung mit $\perp$ auf M

Da man aber meistens VPOs mit Null benötigt, kann man M um ein zusätzliches Element $\perp$ erweitern, wie in Abbildung 3.3 gezeigt.

Wenn man außerdem noch ein größtes Element $\top$ hinzufügt, so erhält man einen vollständigen Verband (Abbildung 3.4).

Definition 3.1.6 (Duale Aussage)
Sei $(P, \perp, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband. Die *duale Aussage* einer gegebenen Aussage erhält man, indem man folgende Symbole miteinander vertauscht: $\sqsubseteq$ mit $\sqsupseteq$, $\sqcup$ mit $\sqcap$, und $\top$ mit $\perp$. □

Wegen der Symmetrie der entsprechenden Definitionen erhält man:

Satz 3.1.1 (Dualitätsprinzip)
Das Duale einer Aussage gilt genau dann, wenn die Aussage selbst gilt. □

Definition 3.1.7 (Monotone Funktion, distributive Funktion)
Eine Funktion $f : D \rightarrow E$ zwischen den zwei partiell geordneten Mengen $(D, \sqsubseteq)$ und $(E, \sqsubseteq)$ heißt *monoton*, wenn

$$\forall d_1, d_2 \in D : d_1 \sqsubseteq d_2 \Rightarrow f(d_1) \sqsubseteq f(d_2)$$

gilt. Sie heißt *distributiv*, wenn

$$\forall d_1, d_2 \in D : f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)$$

ist. □

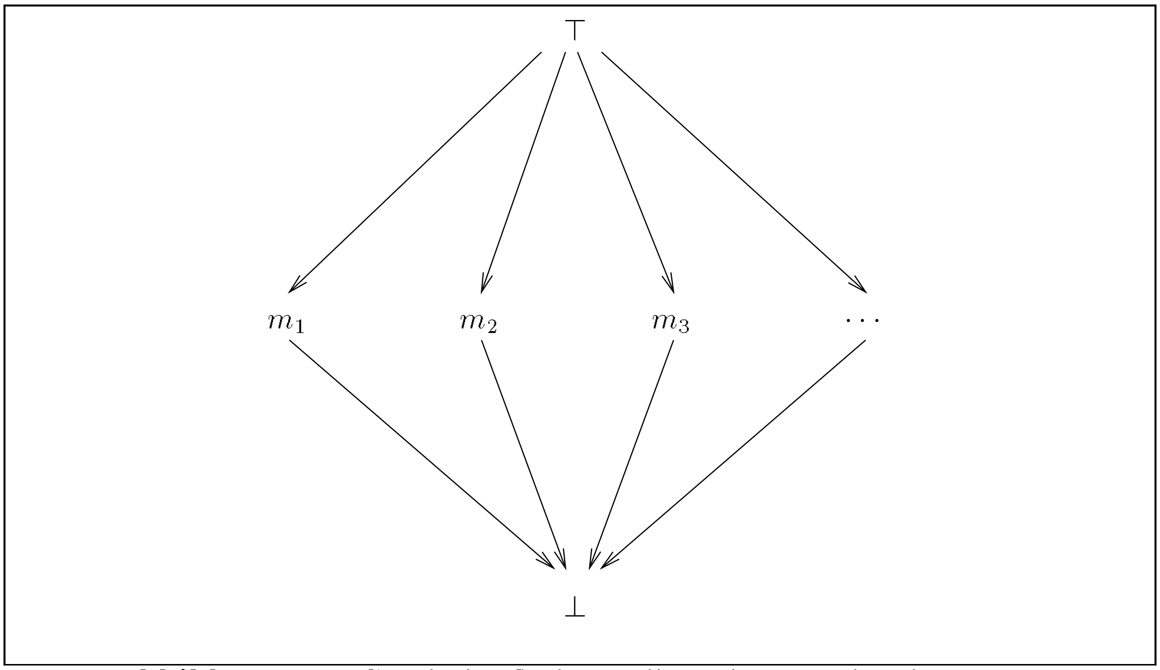

Abbildung 3.4: Graph des flachen vollständigen Verbandes von M

Definition 3.1.8 (Stetige Funktion)
Seien $(D, \sqsubseteq)$ und $(E, \sqsubseteq)$ vollständig partiell geordnete Mengen. Eine Funktion $f : D \rightarrow E$ heißt *stetig* genau dann, wenn für alle Ketten $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_i \sqsubseteq \cdots$ in D gilt

$$\bigsqcup_{n \in \omega} f(d_n) = f(\bigsqcup_{n \in \omega} d_n).$$

□

Satz 3.1.2
Sei $(D, \sqsubseteq)$ eine vollständig partiell geordnete Menge, die die aufsteigende Kettenbedingung erfüllt. Dann ist jede monotone Funktion $f : D \rightarrow D$ auch stetig. □

Beweis 3.1.2
Die obere Schranke einer aufsteigenden Kette in einer partiellen Ordnung, welche die Kettenbedingung erfüllt, ist das Element, das sich wiederholt, wenn die Kette konstant ist.

$$\bigsqcup_{n \in \omega} f(d_n) = f(d_k) \text{ mit } \forall_{i \geq k} d_k = d_i$$

Also gilt:

$$\bigsqcup_{n \in \omega} f(d_n) = f(d_k) = f\left(\bigsqcup_{n \in \omega} d_n\right)$$

mit k groß genug, so daß beide Ketten stabil sind. □

Satz 3.1.3
Sei $(P, \sqsubseteq)$ eine partiell geordnete Menge, in der $\bigsqcap X$ für jede nicht leere Teilmenge $\emptyset \neq X \subseteq P$ existiert. Dann existiert $\bigsqcup X$ in L für jede Teilmenge $X \subseteq L$, die eine obere Schranke in P hat. □

Beweis 3.1.3
Sei X eine beliebige Teilmenge von P, die eine obere Schranke besitzt. Die Menge aller oberen Schranken von X nennen wir X_u, mit $X_u \neq \emptyset$. Dann existiert nach Voraussetzung $\bigsqcap X_u$. Wir zeigen nun, daß $\bigsqcap X_u$ die kleinste obere Schranke von X ist. Sei dazu $x \in X$ beliebig. Nach Definition von X_u ist $x \sqsubseteq u$ für alle $u \in X_u$, d.h. x ist eine untere Schranke von X_u. Daraus folgt $x \sqsubseteq \bigsqcap X_u$. Also ist $\bigsqcap X_u$ eine obere Schranke von X. Betrachte eine weitere obere Schranke m von X. Wegen $m \in X_u$ gilt natürlich $\bigsqcap X_u \sqsubseteq m$. □

Satz 3.1.4
Sei $(P, \sqsubseteq)$ eine nicht leere partiell geordnete Menge. Folgende Aussagen sind äquivalent:

1. P ist ein vollständiger Verband.
2. $\bigsqcap X$ existiert in P für jede Teilmenge $X \subseteq P$.
3. P hat ein größtes Element $\top$ und $\bigsqcap X$ existiert in P für jede nicht leere Teilmenge $X \subseteq P$ □

Beweis 3.1.4

$(1) \Rightarrow (2)$	Folgt aus der Definition eines vollständigen Verbandes.
$(2) \Rightarrow (3)$	Da $\bigsqcap X$ in P für jede Teilmenge $X \subseteq P$ existiert, existiert $\bigsqcap X$ auch für jede nicht leere Teilmenge von P. Wegen $\top = \bigsqcap P$ besitzt P ein größtes Element.
$(3) \Rightarrow (1)$	Sei X eine Teilmenge von P. Da X eine obere Schranke besitzt, nämlich $\top$, und da jede nicht leere Teilmenge von P eine kleinste untere Schranke hat, existiert nach Satz 3.1.3 auch $\bigsqcap X$ in P. □

3.1.2 Konstruktionen auf partiellen Ordnungen und Verbänden

Satz 3.1.5 (Karthesisches Produkt)
Seien $(D_i, \sqsubseteq_i)$ für $i = 1, \cdots, n$ vollständig partiell geordnete Mengen. Dann bildet das Produkt $D_1 \times \cdots \times D_n$ mit

$$(x_1, \cdots, x_n) \sqsubseteq (y_1, \cdots, y_n) :\Leftrightarrow \forall i \in [1..n] : x_i \sqsubseteq_i y_i$$

wieder eine vollständig partiell geordnete Menge. Falls die D_i vollständige Verbände sind, so gilt dies auch für das Produkt. □

Beweis 3.1.5
Zu zeigen ist, daß das Produkt eine vollständig partiell geordnete Menge ist, also jede ω-Kette eine kleinste obere Schranke aus der Produktmenge besitzt. Diese kleinste obere Schranke einer ω-Kette bzw. einer beliebigen Menge wird koordinatenweise berechnet:

$$\bigsqcup_{k\in\omega} (d_{1,k}, \cdots, d_{n,k}) = \left(\bigsqcup_{k\in\omega} d_{1,k}, \cdots, \bigsqcup_{k\in\omega} d_{n,k} \right)$$

Daß dies wirklich die kleinste obere Schranke ist, wird nach der Definition der Ordnung sofort klar. □

Satz 3.1.6
Ein Produktverband erfüllt genau dann die aufsteigende Kettenbedingung, wenn dies alle Komponenten tun. Weiterhin ist die Länge der längsten Kette gleich dem Produkt der Längen der längsten Ketten der Komponenten. □

Beweis 3.1.6
Damit $(d_{1,i}, \cdots, d_{n,i}) \sqsubset (d_{1,i+1}, \cdots, d_{n,i+1})$ ist, muß mindestens für ein $j \in [1..n]$ $d_{j,i} \sqsubset d_{j,i+1}$ sein. Da dies aber wegen der höchstens endlich langen streng aufsteigenden Ketten der Teilverbände nur endlich oft möglich ist, erfüllt auch der Produktverband diese Bedingung. □

Satz 3.1.7 (Potenzverband)
Sei V eine Menge. Dann bildet $(\mathcal{P}(V), \emptyset, V, \subseteq, \cup, \cap)$ einen vollständigen Verband.□

Satz 3.1.8 (Lifting)
Sei $(D, \sqsubseteq_D)$ ein vollständiger Verband. Dann ist $(N, \sqsubseteq_N)$ mit

$$\begin{array}{rcl} N & = & D \cup \{\bot, \top\}^2 \\ n_1 \sqsubseteq_N n_2 & :\Leftrightarrow & \begin{cases} n_1 = \bot \\ n_2 = \top \\ n_1 \sqsubseteq_D n_2 \end{cases} \end{array}$$

ebenfalls ein vollständiger Verband. □

Beweis 3.1.8
Da $(D, \sqsubseteq_D)$ ein vollständiger Verband ist, und $\sqsubseteq_N$ die obigen Eigenschaften besitzt, wird die Verbandseigenschaft nicht verletzt. □

Satz 3.1.9 (Funktionsbildung)
Sei V eine Menge und $(D, \sqsubseteq)$ ein vollständiger Verband. Dann bildet der Funktionenraum der totalen Funktionen von V nach D

$$(V \to D) = \{f \mid f : V \to D\}$$

mit der punktweisen Ordnung

$$f \sqsubseteq g \Leftrightarrow \forall v \in V : f(v) \sqsubseteq g(v)$$

wieder einen vollständigen Verband. $(V \to D)$ bildet eine partiell geordnete Menge bzw. eine VPO, wenn D eine solche ist. □

Beweis 3.1.9
Die kleinste obere Schranke $\hat{f}$ einer Menge von Funktionen $M = \{f_i \mid i \in \omega\}$ existiert, und es gilt

$$\left(\bigsqcup_n f_n\right)(v) = \bigsqcup_n (f_n(v))$$

Daß dies die kleinste obere Schranke ist, wird durch Widerspruch bewiesen: Annahme: $\exists g : g \sqsubset \hat{f}$ und g ist obere Schranke von M
$\Rightarrow \exists v \in V : g(v) \sqsubset \hat{f}(v) = \bigsqcup_n f_n(v)$ Dies ist ein Widerspruch, da g obere Schranke von M ist. Analog zeigt man, daß die kleinste untere Schranke existiert. □

Satz 3.1.10
Seien $(D, \sqsubseteq)$ und $(F, \sqsubseteq)$ mit $F = (V \to D)$ und V endlich, vollständig partiell geordnete Mengen. Dann ist jede streng aufsteigende ω-Kette $f_1 \sqsubset \cdots \sqsubset f_i \sqsubset \cdots$ in F endlich. □

[2] $\bot, \top$ werden so gewählt, daß gilt: $\bot, \top \notin D$

Beweis 3.1.10
Da $|V| = n < \infty$, ist F isomorph zu dem Produkt VPO

$$\underbrace{D \times \cdots \times D}_{n-\text{mal}}$$

der nach Satz 3.1.6 die aufsteigende Kettenbedingung erfüllt. □

3.2 Fixpunkte

Oftmals ist es nötig, rekursive Gleichungssysteme zu lösen, bei denen die zu definierenden Werte nicht nur auf der linken Seite auftauchen. Ein Beispiel ist die Fakultätsfunktion:

$$\text{fac}(n) := \begin{cases} 1 & \text{falls} \quad n = 0 \\ n * \text{fac}(n-1) & \text{sonst} \end{cases}$$

Von einer „Lösung“ wird selbstverständlich erwartet, daß sie die Gleichung erfüllt. Läßt man in diesem Fall allerdings als Lösung partielle Funktionen mit den ganzen Zahlen als Definitionsbereich zu, so erfüllen beliebig viele Funktionen diese Gleichung. Neben den üblichen Fakultätsfunktionen gehört dazu z.B. auch die Funktion

$$f(n) \rightarrow \begin{cases} n! & \text{falls} \quad n \geq 0 \\ -1 & \text{sonst} \end{cases}$$

Aus dieser Vielfalt von Lösungen muß man nun eine auswählen. Dabei hat es sich bewährt, die Funktion zu wählen, die gerade die Definitionsgleichung erfüllt, d.h. eine Funktion, die auf möglichst vielen Elementen undefiniert (bzw. $\perp$) ist. Nach der in Satz 3.1.9 eingeführten Ordnung auf Funktionsräumen ist dies aber gerade die kleinste Funktion der Lösungsmenge.
Diese kann man durch ein natürliches iteratives Verfahren berechnen. Man beginnt mit dem kleinsten Element des Lösungsraumes $\perp_{(Z \rightarrow Z)}$, also der überall undefinierten Funktion. Diese setzt man in die Definitionsgleichung ein und erhält eine größere nur für null definierte Funktion. Diese setzt man wiederum ein und erhält so $\text{fac}^{(2)}$. Nach n-facher Wiederholung dieses Schrittes hat man eine Funktion, die auf dem Intervall $[0..n-1]$ definiert ist und dort die Werte der Fakultätsfunktion annimmt. Für den Grenzfall $n \rightarrow \infty$ erhält man schließlich die gesuchte Funktion auf den natürlichen Zahlen.

Definition 3.2.1 (Fixpunkt, Präfixpunkt)
Sei $f : D \rightarrow D$ eine Funktion. Ein Element $a \in D$ heißt *Fixpunkt* von f, wenn $f(a) = a$. Ein *Präfixpunkt* von f ist ein Element d aus D, so daß $f(d) \sqsubseteq d$. □

Der folgende Satz garantiert bei dem vorgestellten Verfahren, daß eine Lösung existiert.

Satz 3.2.1 (Fixpunkt)
Sei $(D, \sqsubseteq)$ eine vollständig partiell geordnete Menge, $f : X \to X$ eine monotone stetige Funktion. Definiere

$$\mathrm{fix}(f) = \bigsqcup_{n \in \omega} f^n(\perp).$$

Dann ist $\mathrm{fix}(f)$ ein Fixpunkt von f und der kleinste Präfixpunkt von f, d.h.

1. $f(\mathrm{fix}(f)) = \mathrm{fix}(f)$
2. Falls $f(d) \sqsubseteq d$ dann ist $\mathrm{fix}(f) \sqsubseteq d$. □

Beweis 3.2.1

1. Wegen der Stetigkeit von f gilt:

$$\begin{aligned} f\left(\mathrm{fix}(f)\right) &= f\left(\textstyle\bigsqcup_{n\in\omega} f^n(\perp)\right) \\ &= \textstyle\bigsqcup_{n\in\omega} f^{n+1}(\perp) \\ &= \left(\textstyle\bigsqcup_{n\in\omega} f^{n+1}(\perp)\right) \sqcup \{\perp\} \\ &= \textstyle\bigsqcup_{n\in\omega} f^n(\perp) \\ &= \mathrm{fix}(f) \end{aligned}$$

   Also ist $\mathrm{fix}(f)$ ein Fixpunkt von f.

2. Sei d ein Präfixpunkt von f. Selbstverständlich ist $\perp \sqsubseteq d$. Durch die Monotonie folgt $f(\perp) \sqsubseteq f(d)$. Aber d ist ein Präfixpunkt, d.h. $f(d) \sqsubseteq d$, also $f(\perp) \sqsubseteq d$. Und durch Induktion folgt $f^n(\perp) \sqsubseteq d$. So ist $\mathrm{fix}(f) = \bigsqcup_{n\in\omega} f^n(\perp) \sqsubseteq d$.
   Da jeder Fixpunkt ein Präfixpunkt ist, ist $\mathrm{fix}(f)$ der kleinste Fixpunkt von f. □

3.3 Abstrakte Interpretation

Nachfolgend sollen die theoretischen Grundlagen der abstrakten Interpretation untersucht werden. Diese sind auch für die Datenflußanalyse von besonderer Bedeutung, da man sie, wie in diesem Kapitel gezeigt wird, als Spezialfall der abstrakten Interpretation auffassen kann.

Die Datenflußanalyse besitzt operationalen Charakter. Daher geht sie aus einer abstrakten Interpretation einer operationalen Semantik hervor. Grundlage einer solchen Semantik sind eine mathematische Maschine und ihre Berechnungen.

Definition 3.3.1 (Mathematische Maschine)
Eine (nichtdeterministische) *mathematische Maschine* ist ein vier-Tupel $(Z, A, E, \rightarrow)$, bestehend aus einer Menge Z von Zuständen, einer Menge $A \subseteq Z$ von Anfangszuständen, einer Menge $E \subseteq Z$ von Endzuständen und einer Übergangsrelation $\rightarrow \subseteq (Z \backslash E) \times Z$.
Ein *Berechnungsschritt* ist ein Übergang $z \rightarrow z'$ mit $z, z' \in Z$. Eine Folge von Berechnungsschritten $z_0 \rightarrow z_1 \rightarrow \cdots$ mit $z_0 \in A$ heißt *Berechnung*. Eine Berechnung $z_0 \rightarrow \cdots \rightarrow z_n$ *terminiert*, wenn es keinen Übergang $z_n \rightarrow z$ gibt. Falls $z_n \in E$ ist, heißt die Berechnung *fehlerfrei*, sonst *fehlerhaft*.
Mit $\rightarrow^*$ bezeichnet man die reflexive, transitive Hülle von $\rightarrow$. □

Definition 3.3.2 (Erreichbare Zustände)
Ist $M = (Z, A, E, \rightarrow)$ eine mathematische Maschine und $Z_0 \subseteq Z$ eine Menge von Zuständen, dann heißt die kleinste Teilmenge von $Z, E_M(Z_0)$ mit den Eigenschaften

- $Z_0 \subseteq E_M(Z_0)$
- $\forall z \in E_M(Z_0), z' \in Z : z \rightarrow z' \Rightarrow z' \in E_M(Z_0)$

die *Menge der von Z_0 in M erreichbaren Zustände*. $E_M(Z_0)$ ist also die kleinste gegenüber $\rightarrow$ abgeschlossene Obermenge von Z_0. $z \in E_M(Z_0)$ heißt *von Z_0 erreichbar*. □

Definition 3.3.3 (Programmpunkt)
Ein *Programmpunkt* wird dargestellt als ein Programmtext, in den ein $\bullet$ vor oder hinter einer Anweisung eingefügt wird. Mit $\mathcal{P}$ wird die Menge aller Programmpunkte bezeichnet. Programmpunkte werden meist in Verbindung mit variablen Programmteilen α, β, γ benutzt, z.B. $[\alpha \bullet c\gamma]$. □

Nichtterminale				
	A, E, P			
Startsymbol				
	P			Programm
Externe Menge				
	$v \in \mathcal{V}$			Variablen
Regeln				
	P	$\rightarrow$	A	
	A	$\rightarrow$	$v := E$	Zuweisung
		|	**if** E **then** A **else** A **fi**	
		|	**while** E **do** A **od**	
		|	(A;A)	Hintereinanderausführung
	E	$\rightarrow$	$1 \mid 2 \mid \cdots$	
		|	v	
		|	E + E	
		|	E * E	

Abbildung 3.5: Die kontextfreie Grammatik einer einfachen imperativen Sprache

Als Beispiel soll hier die in Abbildung 3.6 definierte mathematische Maschine M_1 dienen, welche die Standardsemantik der imperativen Programmiersprache aus Abbildung 3.5 darstellt.

Man ist interessiert an Aussagen über alle Zustände, in denen die Kontrolle vor einer Anweisung A steht und die von den Startzuständen aus erreichbar sind, z.B. „Die Variable x hat vor der Anweisung A immer den Wert 3“. Dies kann formuliert werden als $\forall (p, vb) \in E_M(\bullet P, \lambda v.0) : p = [\alpha \bullet A\beta] \Rightarrow vb(x) = 3$

Nun kann eine andere Maschine konstruiert werden, die M_1 simuliert, und auf der diese Aussage ebenfalls ausgedrückt werden kann. Diese Art der Simulation ist durch die abstrakte Interpretation gegeben.

Definition 3.3.4 (Abstrakte Interpretation)
Seien $M = (Z, A, E, \rightarrow)$ und $\overline{M} = (\overline{Z}, \overline{A}, \overline{E}, \overline{\Rightarrow})$ mathematische Maschinen. $(\overline{M}, \delta)$ mit $\delta \in Z \times \overline{Z}$ ist eine *abstrakte Interpretation* von M, wenn gilt:

1. $\forall z \in A : \exists \overline{z} \in \overline{A} : z\, \delta\, \overline{z}$
   d.h. für jeden Anfangszustand in M gibt es eine abstrakte Repräsentation in $\overline{M}$, und

2. $\forall z, z' \in Z, \overline{z} \in \overline{Z} : z\, \delta\, \overline{z} \wedge z \rightarrow z' \Rightarrow \exists \overline{z}' \in \overline{Z} : z'\, \delta\, \overline{z}' \wedge \overline{z} \overline{\Rightarrow}^{*} \overline{z}'$
   d.h. für jeden Berechnungsschritt in M gibt es eine korrekte Abstraktion in $\overline{M}$. □

- Variablenbindung:
  $VB = \mathcal{V} \to \mathbb{Z}$ Abbildung von Variablen auf Werte
- Zustände:
  $Z = \mathcal{P} \times VB$ Paar aus Programmpunkt und aktueller Variablenbindung
- Anfangszustände:
  $A = \{([\bullet\alpha], \lambda v.0) \in \mathcal{P} \times VB\}$
- Endzustände:
  $E = \{([\alpha\bullet], vb) \in \mathcal{P} \times VB\}$
- Übergänge (für alle $\alpha, \beta, \gamma, vb, E, A, v, \cdots$):
  Sei $[\![\ ]\!]$ die Standardsemantik für Ausdrücke
  $([\alpha \bullet v := E\gamma], vb) \to ([\alpha v := E \bullet \gamma], vb[v \mapsto [\![E]\!](vb)])$

  $([\alpha \bullet \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \gamma], vb)$
  $\to ([\alpha \text{ if } E \text{ then } \bullet \beta_1 \text{ else } \beta_2 \text{ end } \gamma], vb)$, falls $[\![E]\!](vb) \neq 0$
  $([\alpha \bullet \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \gamma], vb)$
  $\to ([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \bullet \beta_2 \text{ end } \gamma], vb)$, falls $[\![E]\!](vb) = 0$
  $([\alpha \text{ if } E \text{ then } \beta_1 \bullet \text{ else } \beta_2 \text{ end } \gamma], vb)$
  $\to ([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \bullet \gamma], vb)$
  $([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \bullet \text{ end } \gamma], vb)$
  $\to ([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2) \text{ end } \bullet \gamma], vb)$

  $([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], vb)$
  $\to ([\alpha \text{ while } E \text{ do } \bullet \beta \text{ od } \gamma], vb)$, falls $[\![E]\!](vb) \neq 0$
  $([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], vb)$
  $\to ([\alpha \text{ while } E \text{ do } \beta \text{ od } \bullet \gamma], vb)$, falls $[\![E]\!](vb) = 0$
  $([\alpha \text{ while } E \text{ do } \beta \bullet \text{ od } \gamma], vb)$
  $\to ([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], vb)$

  $([\alpha \bullet (A_1; A_2)\beta], vb) \to ([\alpha(\bullet A_1; A_2)\beta], vb)$
  $([\alpha(A_1\bullet; A_2)\beta], vb) \to ([\alpha(A_1; \bullet A_2)\beta], vb)$
  $([\alpha(A_1; A_2\bullet)\beta], vb) \to ([\alpha(A_1; A_2) \bullet \beta], vb)$

Abbildung 3.6: Operationale Semantik für die Sprache aus Abbildung 3.5

- Zustände:
  $\overline{Z} = \mathcal{P} \to V$ punktweise geordnet und nach Satz 3.1.9 ein vollständiger Verband.
- Abstraktionsrelation:
  $(p, v)\ \delta\ \pi :\Leftrightarrow v \sqsubseteq \pi(p)$
- Anfangszustände: $\overline{A} = \overline{Z}$
- Endzustände: Alle Zustände, für die es keinen Übergang gibt.
- Übergänge:
  $\pi \overline{\rightarrow} \pi' :\Leftrightarrow \pi \sqsubset \pi' \land \exists\, (p, \pi(p)) \to (p', v') : \pi' = \pi[p' \mapsto \pi(p') \sqcup v']$
  D.h. es gibt einen Übergang aus π, wenn es einen Programmpunkt p gibt, für den es in der ursprünglichen Maschine einen Übergang unter $\pi(p)$ zu einem Paar (p', v') gibt, und dieses v' echt größer ist als die unter p' in π abgelegte Information. Der neue Zustand durch die Abschwächung der Information an p' in π durch v'.

Abbildung 3.7: Konstruktion einer kumulierenden abstrakten Interpretation $(M_{kum(M)}, \delta)$ mit $M_{kum(M)} = (\overline{Z}, \overline{A}, \overline{E}, \overline{\rightarrow})$ für eine Maschine $M = (Z, A, E, \to)$, deren Zustände die Form $Z = \mathcal{P} \times V$ haben, wobei $(V, \sqsubseteq)$ einen vollständigen Verband bildet.

Als Beispiel für eine abstrakte Interpretation zu einer Maschine M, deren Zustände Z die Form $\mathcal{P} \times V$ haben, wobei $(V, \sqsubseteq)$ einen vollständigen Verband bilden muß, sei hier die Maschine $M_{kum(M)}$ genannt, die wie in Abbildung 3.7 beschrieben, aus M konstruiert wird.

Satz 3.3.1
Sei $M = (Z, A, E, \to)$ eine mathematische Maschine mit $Z = \mathcal{P} \times V$ und $(V, \sqsubseteq)$ ein vollständiger Verband. Dann ist das in Abbildung 3.7 definierte Paar $(M_{kum(M)}, \delta)$ eine abstrakte Interpretation von M. $\square$

Beweis 3.3.1
Zu beweisen sind die beiden in Definition 3.3.4 genannten Punkte:

1. Sei $(p, v) \in A$ beliebig. Sei
$$\pi = \lambda x. \begin{cases} v & \text{falls } x = p \\ \perp & \text{sonst} \end{cases}$$
   Dann gilt $(p, v)\, \delta\, \pi$ und außerdem $\pi \in \overline{A}$.

2. Seien $z = (p, v), z' = (p', v') \in Z$ und $\pi \in \overline{Z}$ beliebig, mit $z \rightarrow z'$ und $z\,\delta\,\pi$. Sei $\pi' = \pi[p' \mapsto \pi(p') \sqcup v']$. Da $\pi(p') \sqsubseteq \pi(p') \sqcup v'$ ist, gilt $\pi \sqsubseteq \pi'$. Entweder ist $\pi \sqsubset \pi'$, dann gibt es einen Übergang $\pi \Rightarrow \pi'$ oder $\pi = \pi'$, dann gilt aber $\pi \Rightarrow^* \pi'$.
   Bleibt zu zeigen, daß $z'\,\delta\,\pi'$ gilt. Dies ist aber klar, da $v' \sqsubseteq \pi'(p')$ ist. □

Man kann nun zeigen, daß es für jeden in einer Maschine M erreichbaren Zustand z eine Abstraktion in einer abstrakten Interpretation $(\overline{M}, \delta)$ gibt, die ebenfalls erreichbar ist.

Satz 3.3.2
Ist $M = (Z, A, E, \rightarrow)$ eine mathematische Maschine, $(\overline{M}, \delta)$ mit $\overline{M} = (\overline{Z}, \overline{A}, \overline{E}, \Rightarrow)$ eine abstrakte Interpretation von M, dann gilt für $Z' \subseteq Z$ und $\overline{Z}' \subseteq \overline{Z}$:

$$\left(\forall z' \in Z' : \exists \overline{z}' \in \overline{Z}' : z'\,\delta\,\overline{z}'\right) \quad \Rightarrow \quad \left(\forall z' \in E_M(Z') : \exists \overline{z}' \in E_{\overline{M}}(\overline{Z}') : z'\,\delta\,\overline{z}'\right)$$

oder[3]

$$Z' \subseteq \cdot\,\delta\,\overline{Z}' \quad \Rightarrow \quad E_M(Z') \subseteq \cdot\,\delta\,E_{\overline{M}}(\overline{Z}')$$

□

Beweis 3.3.2
Seien $Z', \overline{Z}'$ mit $\forall z' \in Z' : \exists \overline{z}' \in \overline{Z}' : z'\,\delta\,\overline{z}'$ beliebig.
Sei $z \in E_M(Z')$, d.h. $\exists z_0 \rightarrow \cdots \rightarrow z_n : z_0 \in Z' \wedge z_n = z$. Da $z_0 \in Z'$ ist, existiert nach Voraussetzung ein $\overline{z_0} \in \overline{Z}'$ mit $z_0\,\delta\,\overline{z_0}$.
Nach induktiver Erweiterung der Definition der abstrakten Interpretation (3.3.4) existiert dann ein $\overline{z} \in \overline{Z}$ mit $z\,\delta\,\overline{z}$ und $\overline{z_0} \Rightarrow^* \overline{z}$. Diese letzte Aussage heißt aber gerade, daß $\overline{z} \in E_{\overline{M}}(\overline{Z}')$ ist. □

Man versucht, für die betrachtete Aussage P über einen Zustand in der Originalmaschine $M = (Z, A, E, \rightarrow)$ eine Aussage $\overline{P}$ über einen Zustand der abstrakten Interpretation $(\overline{M}, \delta)$ mit $\overline{M} = (\overline{Z}, \overline{A}, \overline{E}, \Rightarrow)$ zu finden, für die geschlossen werden kann, daß $P(z)$ gilt, wenn $\overline{P}(\overline{z})$ wahr ist für einen Zustand $\overline{z}$, der z repräsentiert. Dann gilt, daß P für alle in M erreichbaren Zustände wahr ist, falls die Berechnung in der Maschine $\overline{M}$ ergab, daß $\overline{P}$ für alle erreichbaren Zustände galt. Dies formalisiert der nachstehende Satz.

Satz 3.3.3
Seien $(\overline{M}, \delta)$ mit $\overline{M} = (\overline{Z}, \overline{A}, \overline{E}, \Rightarrow)$ eine abstrakte Interpretation von $M = (Z, A, E,$

[3] Die Notation $\cdot\,\delta\,\overline{Z}$ bezeichnet die Menge aller z, die durch die $\overline{z}$ in $\overline{Z}$ repräsentiert werden. Also $\cdot\,\delta\,\overline{Z} = \{z \mid z\,\delta\,\overline{z} \text{ mit } \overline{z} \in \overline{Z}\}$

$\rightarrow$), und $Z_0 \subseteq Z, \overline{Z_0} \subseteq \overline{Z}$ mit $Z_0 \,\delta\, \overline{Z_0}$[4]. Sei weiterhin P ein einstelliges Prädikat über Z und $\overline{P}$ ein solches über $\overline{Z}$. Falls gilt

$$\forall z \in Z, \overline{z} \in \overline{Z} : z \,\delta\, \overline{z} \wedge \left(\overline{P}(\overline{z}) \Rightarrow P(z)\right)$$

so folgt

$$\left(\forall \overline{z} \in E_{\overline{M}}(\overline{Z_0}) : \overline{P}(\overline{z})\right) \Rightarrow (\forall z \in E_M(Z_0) : P(z))$$

□

Beweis 3.3.3
Gelte $\forall \overline{z} \in E_{\overline{M}}(\overline{Z_0}) : \overline{P}(\overline{z})$, und sei $z \in E_M(Z_0)$ beliebig.
Dann gibt es nach Satz 3.3.2 ein $\overline{z} \in E_{\overline{M}}(\overline{Z_0})$ mit $z \,\delta\, \overline{z}$. Nach Voraussetzung gilt dafür gerade $\overline{P}(\overline{z})$, woraus mit der Bedingung für $\overline{P}$ folgt, daß $P(z)$ gilt. □

So können Maschinen mit interessanten Eigenschaften konstruiert werden, mittels derer wahre Aussagen über die Originalmaschine gemacht werden können. Wünschenswerte Eigenschaften sind dabei Terminierung auf allen legalen Startzuständen, die Berechenbarkeit der Übergänge und die Quantifizierung der Aussage über endliche Mengen. Der letzte Punkt führt dazu, daß die potentiell unendliche Menge der erreichbaren Zustände durch endliche Mengen abstrahiert werden muß.

Durch die Forderung der sogenannten Kumulationseigenschaft für die abstrakten Interpretationen erreicht man, daß der letzte Zustand in einer terminierenden Berechnung von $\overline{M}$ alle von einer Zustandsmenge Z_0 aus in M erreichbaren Zustände repräsentiert, falls die Rechnung von $\overline{M}$ mit einem Zustand $\overline{z_0}$ begonnen hat, der Z_0 abstrahiert.
Diese Eigenschaft besagt nämlich daß, falls ein Übergang $\overline{z} \overline{\rightarrow} \overline{z}'$ existiert, $\overline{z}'$ eine Obermenge der Zustände repräsentieren muß, die $\overline{z}$ repräsentiert.

Definition 3.3.5 (Kumulationseigenschaft)
Eine mathematische Maschine $M = (Z, A, E, \rightarrow)$ besitzt die Kumulationseigenschaft, wenn gilt $\forall z, z' \in Z : z \rightarrow z' \Rightarrow z \sqsubset z'$. □

Satz 3.3.4
Sei $M = (Z, A, E, \rightarrow)$ eine mathematische Maschine und $(\overline{M}, \delta)$ eine abstrakte Interpretation dazu, die die Kumulationseigenschaft erfüllt. Seien ferner $\overline{z_0} \overline{\rightarrow} \cdots \overline{\rightarrow} \overline{z_n}$ eine terminierte Berechnung von $\overline{M}$ und $Z_0 \subseteq Z$. Falls $\forall z \in Z_0 : z \,\delta\, \overline{z_0}$, dann gilt $\forall z \in E_M(Z_0) : z \,\delta\, \overline{z_n}$. □

Beweis 3.3.4
Sei $z \in E_M(Z_0)$ beliebig. Es gibt also eine Berechnung $z_0 \rightarrow \cdots \rightarrow z_n$ von M.

[4] d.h. $\forall z_0 \in Z_0 : \exists \overline{z_0} \in \overline{Z_0} : z_0 \,\delta\, \overline{z_0}$.

1. Wähle einen Programmpunkt p, der nicht markiert ist.
2. Falls es keinen solchen gibt, beende die Berechnung.
3. Berechne einen Übergang $(p, \pi(p)) \rightarrow (p', \overline{v}')$.
4. Falls es keinen solchen gibt, markiere p und fahre bei 1 fort.
5. Bilde $x = \pi(p') \sqcup \overline{v}'$.
6. Teste, ob $\pi(p') \sqsubset x$.
7. Falls ja, brich mit dem Ergebnis $\pi[p' \mapsto x]$ ab.
8. Falls nein, markiere p und fahre bei 1 fort.

Abbildung 3.8: Algorithmus zur Berechnung eines Folgezustandes π' aus π für eine Maschine $M_{kum(M)}$

Dann existiert nach Definition der abstrakten Interpretation (3.3.4) eine Berechnung $\overline{z_0} \Rightarrow \overline{z_1'} \Rightarrow \cdots \Rightarrow \overline{z_k'}$ mit $z_n \,\delta\, \overline{z_k'}$. Sei nun i der minimale Index, für den gilt: $\overline{z_i} \neq \overline{z_i'}$. Aber $\overline{z_i'}$ ist größer als $\overline{z_{i-1}}$, also repräsentieren alle $\overline{z_j}$ mit $j > i$ ebenfalls den Zustand von M, für den der unterschiedliche Übergang gemacht wurde. Da die Berechnung $\overline{z_0} \Rightarrow \cdots \Rightarrow \overline{z_n}$ aber terminiert ist, muß der entsprechende Übergang bereits gemacht worden sein bevor $\overline{z_n}$ erreicht wird. Daher gilt: $z \,\delta\, \overline{z_n}$. □

In einer Maschine mit Kumulationseigenschaft ist jeder Zustand in einer Berechnung kleiner als sein Folgezustand. Wenn nun aber die partiell geordnete Menge $(Z, \sqsubseteq)$ die aufsteigende Kettenbedingung erfüllt, so ist jede Berechnung endlich, terminiert also insbesondere für alle legalen Startzustände. Wurde die oben vorgeschlagene Konstruktion für kumulierende Maschinen benutzt, so ist die Zustandsmenge die Menge der Funktion von Programmpunkten in eine partiell geordnete Menge $(V, \sqsubseteq)$ von Werten und ist punktweise geordnet. Da aber die Menge der für ein Programm P existierenden Programmpunkte endlich ist, es also nur begrenzt viele Programmpunkte gibt, erfüllt $(Z, \sqsubseteq)$ nach Satz 3.1.10 genau dann die aufsteigende Kettenbedingung, wenn $(V, \sqsubseteq)$ diese erfüllt.

Die Terminierung von $M_{kum(M)}$ kann also garantiert werden, falls $(V, \sqsubseteq)$ die aufsteigende Kettenbedingung erfüllt.

Ist die abstrakte Maschine $\overline{M}$ so konstruiert, daß sie die Kumulationseigenschaft besitzt, ihre Übergangsrelation berechenbar ist und sie zusätzlich für jeden legalen Anfangszustand terminiert, so kann man mit einem $\overline{z_0}$ mit $Z_0 \,\delta\, \overline{z_0}$ die Repräsentation von $E_M(Z_0)$ berechnen.

Wurde die kumulierende Maschine M_{kum} als Grundlage gewählt, so stellt sich die Berechnung eines π', für das gilt $\pi \to \pi'$ wie in Algorithmus 3.8 dar.

Allerdings terminiert die Berechnung der kumulierenden Maschine nicht im allgemeinen, da der zugrunde liegende vollständige Verband $(Z, \sqsubseteq)$ nicht die aufsteigende Kettenbedingung erfüllt. Daher abstrahiert man zuerst durch $(\overline{M}, \delta_1)$ von der Orginalmaschine M, so daß $\overline{M}$ auf allen interessanten Zuständen terminiert. Anschließend bildet man $(M_{kum(\overline{M})}, \delta_2)$ von $\overline{M}$, um die Allquantifizierung der untersuchten Aussage zu eliminieren.

Die abstrakte Interpretation $(\overline{M}, \delta)$ für das Beispiel der Konstantenpropagation ist in Abbildung 3.9 zu sehen. Aus Abbildung 3.4 und der Tatsache, daß es nur endlich viele Variablen in einem Programm gibt, folgt mit Satz 3.1.10, daß der $\overline{M}$ zugrunde liegende Verband die aufsteigende Kettenbedingung erfüllt und $\overline{M}$ sowie $M_{kum(\overline{M})}$ folglich terminieren.

- Abstrakte Variablenbindung:
  $\overline{VB} = \mathcal{V} \to \overline{\mathbb{Z}}$ Abbildung von Variablen auf Werte (vgl. Abbildung 3.4)
- Zustand:
  $\overline{Z} = \mathcal{P} \times \overline{VB}$ Paar aus Programmpunkt und aktueller Variablenbindung
- Abstraktionsrelation: $(p_1, vb)\,\delta\,(p_2, \overline{vb}) \Leftrightarrow p_1 = p_2 \wedge \forall v \in \mathcal{V} : vb(v)\,\delta_Z\,\overline{vb}(v)$, wobei $z\,\delta_Z\,\overline{z} \Leftrightarrow z = \overline{z} \vee \overline{z} = \top$
- Anfangszustände: $\overline{A} = \{([\bullet\alpha], \overline{vb}) \in \mathcal{P} \times \overline{VB}\}$
- Endzustände:
  $\overline{E} = \{([\alpha\bullet], \overline{vb}) \in \mathcal{P} \times \overline{VB}\}$
- Übergänge (für alle $\alpha, \beta, \gamma, \overline{vb}, E, v, \cdots$):
  Sei $[\![\ ]\!]$ die Standardsemantik für Ausdrücke erweitert auf $\overline{Z}$
  $\big([\alpha \bullet v := E\gamma], \overline{vb}\big) \to \big([\alpha v := E \bullet \gamma], \overline{vb}[v \mapsto [\![E]\!](\overline{vb})]\big)$

  $\big([\alpha \bullet \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \gamma], \overline{vb}\big)$
  $\to \big([\alpha \text{ if } E \text{ then } \bullet \beta_1 \text{ else } \beta_2 \text{ end } \gamma], \overline{vb}\big)$,
  $\big([\alpha \bullet \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \gamma], \overline{vb}\big)$
  $\to \big([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \bullet \beta_2 \text{ end } \gamma], \overline{vb}\big)$,
  $\big([\alpha \text{ if } E \text{ then } \beta_1 \bullet \text{ else } \beta_2 \text{ end } \gamma], \overline{vb}\big)$
  $\to \big([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \text{ end } \bullet \gamma], \overline{vb}\big)$
  $\big([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2 \bullet \text{ end } \gamma], \overline{vb}\big)$
  $\to \big([\alpha \text{ if } E \text{ then } \beta_1 \text{ else } \beta_2) \text{ end } \bullet \gamma], \overline{vb}\big)$

  $\big([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], \overline{vb}\big) \to \big([\alpha \text{ while } E \text{ do } \bullet \beta \text{ od } \gamma], \overline{vb}\big)$
  $\big([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], \overline{vb}\big) \to \big([\alpha \text{ while } E \text{ do } \beta \text{ od } \bullet \gamma], \overline{vb}\big)$
  $\big([\alpha \text{ while } E \text{ do } \beta \bullet \text{ od } \gamma], \overline{vb}\big) \to \big([\alpha \bullet \text{ while } E \text{ do } \beta \text{ od } \gamma], \overline{vb}\big)$

  $([\alpha \bullet (A_1; A_2)\beta], vb) \to ([\alpha(\bullet A_1; A_2)\beta], vb)$
  $([\alpha(A_1\bullet; A_2)\beta], vb) \to ([\alpha(A_1; \bullet A_2)\beta], vb)$
  $([\alpha(A_1; A_2\bullet)\beta], vb) \to ([\alpha(A_1; A_2) \bullet \beta], vb)$

Abbildung 3.9: Abstrakte Interpretation der Semantik aus Abbildung 3.6

Zusammenfassung

Problem:
Für eine Maschine $M = (\mathcal{P} \times V, A, E, \rightarrow)$ und eine Aussage $P(Z_0)$ der Form $\forall (p,v) \in E_M(Z_0) : Q\left((p, vb)\right)$ soll ein berechenbares Teilprädikat $P'(Z_0)$[5] geschaffen werden.
Lösung:

1. Es wird eine abstrakte Maschine $\overline{M} = (\mathcal{P} \times \overline{V}, \overline{A}, \overline{E}, \overline{\Rightarrow})$ mit der Abstraktionsrelation δ_1 gesucht, wobei $(p_1, v)\, \delta_1\, (p_2, \overline{v}) \Leftrightarrow p_1 = p_2 \wedge v\, \delta_1'\, \overline{v}$ ist. Außerdem muß $(\overline{V}, \sqsubseteq)$ ein vollständiger Verband sein, der die aufsteigende Kettenbedingung erfüllt und dessen Ordnung den Informationsgehalt widerspiegelt.
   Q wird transformiert in eine Aussage $\overline{Q}$, für die gilt: $(p,v)\, \delta_1\, (p, \overline{v}) \wedge \overline{Q}\left((p, \overline{v})\right) \Rightarrow Q\left((p,v)\right)$. Außerdem wird eine berechenbare Funktion f_1 benötigt, die zulässige Startzustände von M auf Startzustände von $\overline{M}$ abbildet.

2. Dann wird die Maschine $M' = M_{kum(\overline{M})}$ konstruiert und eine Funktion f_2 gesucht, die ein $\overline{Z_0}$ auf ein π_0' mit $\overline{Z_0}\, \delta\, \pi_0'$ abbildet. $\overline{P}$ wird in die Aussage $P'(\pi_0') = \exists$ terminierte Berechnung $\pi_0' \rightarrow'^{*} \pi_n' : Q'(\pi)$ mit $Q'(\pi) = \forall p : \forall \overline{vb} : \overline{Q}\left((p, \overline{v})\right)$ transformiert.
   Dann gilt: $P'\left(f_2\left(f_1^{*}\left(Z_0\right)\right)\right) \Rightarrow P(Z_0)$[6].

Algorithmus:
Eingabe: Eine Zustandsmenge Z_0, für die geprüft werden soll, ob $P(Z_0)$ gilt.

1. Berechne mittels f_1 und f_2 eine Repräsentation π_0' für Z_0

2. Führe eine Berechnung der Maschine M' auf π_0' aus, deren letzter Zustand π_n' sei (wie in Bild 3.8 beschrieben).

3. Prüfe, ob P' für π_n' gilt, und gib die Antwort aus.

Fazit

Dieses Verfahren entspricht genau einer Datenflußanalyse. Man hat also die Datenflußanalyse als abstrakte Interpretation dargestellt und so einen direkten Zusammenhang der konkreten (operationalen) Semantik einer Programmiersprache zur Datenflußanalyse.

So ist es möglich, für Datenflußanalysen die Korrektheit in bezug auf eine konkrete Semantik nachzuweisen.

[5] d.h. $\{Z \mid P'(Z)\} \subseteq \{Z \mid P(Z)\}$

[6] Mit f^{*} wird die mengenwertige Erweiterung von f bezeichnet, d.h. $f^{*}(X) = \{f(x) \mid x \in X\}$.

Kapitel 4

Datenflußanalyse

Die heutzutage praktizierte Datenflußanalyse wurde hauptsächlich von Kildall in [Kil73] eingeführt und von Kam in [KU77] verfeinert. Einen guten Überblick über die gesamte Theorie gibt Zima in seinem Buch [ZC91].

4.1 Grundlegendes

Definition 4.1.1 (Kontrollflußgraph (KFG))
Ein Viertupel $G = (N, E, s, e)$ mit einer Menge N von Knoten, $E \subseteq N \times N$ einer Menge von gerichteten Kanten, einem Startknoten $s \in N$ und einem Endknoten $e \in N$ (d.h. s besitzt keine eingehende und n keine ausgehende Kante) heißt *Kontrollflußgraph.* Zusätzlich muß eine Beschriftungsfunktion $F_G : N \rightarrow \mathcal{S}$ existieren, die den Knoten Programmstücke zuordnet. Dabei ist $\mathcal{S}$ die Menge der Syntaxbaumfragmente der zugrunde liegenden Sprache. □

Die Konstruktion dieser Graphen wird z.B. in [All70] beschrieben. Daß die obige Definition einen Start- und Endknoten verlangt, ist keine Einschränkung, da für jeden anderen Graphen $G = (N, E)$ mit Mengen von Eintritts- und Austrittsknoten A bzw. B ein Graph $G' = (N \cup \{s, e\}, E', s, e)$ mit $E' = E \cup \{(s, v) \mid v \in A\} \cup \{(v, e) \mid v \in B\}$ konstruiert werden kann, der diese Bedingung erfüllt.

Ein solcher Kontrollflußgraph repräsentiert die möglichen Ausführungswege durch ein Programm P. Da diese durch abstrakte Syntaxbäume nicht ausreichend modelliert werden können, wurde ein Graph als Grundlage der Analyse gewählt.

Definition 4.1.2 (Pfad)
Sei $G = (N, E, s, e)$ ein Kontrollflußgraph. Eine Folge $\pi = n_1, n_2, \cdots, n_k$ von Knoten heißt ein *Pfad* von n_1 nach n_k durch G, falls gilt: $\forall i \in [1..k-1] : (n_i, n_{i+1}) \in E$. Mit ϵ wird der leere Pfad bezeichnet. $\mathbf{P}[n_1, n_2]$ ist die Menge aller Pfade von n_1 nach n_2. Weiterhin bezeichnet ';' die Konkatenation zweier Pfade. □

Definition 4.1.3 (Datenflußproblem (DFP))
Sei $V = (A, \perp, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband[1]. $D = (G, V, [\![\ ]\!])$ heißt *Datenflußproblem*, wenn $G = (N, E, s, e)$ ein Kontrollflußgraph und $[\![\ ]\!] : N \rightarrow F$ eine Funktion ist, die den Knoten des Flußgraphen Elemente aus F zuordnet, die *Transferfunktionen* oder *Übergangsfunktionen* genannt werden. □

Definition 4.1.4 (Monotone und distributive DFP)
Sei $D = (G, V, [\![\ ]\!])$ ein Datenflußproblem. D heißt *monoton*, falls für alle Knoten n von G die Funktion $[\![n]\!]$ monoton ist. D heißt *distributiv*, falls alle $[\![n]\!]$ distributiv sind. □

Definition 4.1.5 (Pfadsemantik)
Sei $D = (G, V, [\![\ ]\!])$ ein Datenflußproblem. Die semantische Funktion $[\![\ ]\!]$ für Knoten n läßt sich wie folgt auf einen Pfad $\pi = n_1, \cdots, n_k \in \mathbf{P}[n_1, n_k]$ erweitern:

$$[\![\pi]\!] = \begin{cases} id & \text{falls } \pi = \epsilon \\ [\![(n_2, \cdots, n_k)]\!] \circ [\![n_1]\!] & \text{sonst} \end{cases}$$

Dabei bezeichnet $\circ$ die Funktionskomposition. □

Es ist leicht einzusehen, daß für ein monotones Datenflußproblem $[\![\pi]\!]$ für alle Pfade π ebenfalls monoton ist. Ebenso ist $[\![\pi]\!]$ distributiv, falls das DFP distributiv ist.

Die gesuchte Lösung eines Datenflußproblemes D für einen Knoten n ist die Vereinigung aller Pfadsemantiken für Pfade von s nach n, angewandt auf den Startwert $\perp$[2].

[1] es ist auch möglich, mit einer partiellen Ordnung $(P, \sqsubseteq)$ zu arbeiten, fall diese für jede Teilmenge von P eine kleinste obere Schranke besitzt. (Eine solche Ordnung wird auch Vereinigungshabverband genannt).

[2] Zum Teil werden auch andere Startwerte $v_s \in V$ verwendet, dies läßt sich aber äquivalent formulieren, indem man die semantische Funktion für den Startknoten durch die konstante Funktion $[\![s]\!](v) =_{def} [\![s]\!](v_s)$ ersetzt.

Definition 4.1.6 (MOP-Lösung)
Sei $D = (G, V, [\![\ ]\!])$ mit $G = (N, E, s, e)$ ein Datenflußproblem. Die meet-over-all-paths-Lösung[3] wird für alle $n \in N$ definiert als

$$\mathbf{MOP}(n) = \bigsqcup\{[\![\pi]\!](\perp) | \pi \in \mathbf{P}[s, n]\}$$

□

Bei der obigen Definition ist zu beachten, daß die Menge der Pfade vom Startknoten zu einem Knoten n im allgemeinen eine unendliche Mächtigkeit haben kann. Deshalb kann die Definition nicht als Berechnungsvorschrift verstanden werden. Es wird also eine algorithmische Lösungsstrategie gesucht, die eine Approximation der gesuchten Lösung berechnet. Dabei heißt eine Approximation korrekt, falls sie kleiner oder gleich der gesuchten Lösung bzgl. der Ordnung des benutzten Verbandes ist. Die **MOP**-Lösung wird auch **präzise** genannt, da es die genaueste aller Approximationen ist. Eine (algorithmische) Lösungsstrategie ist die maximum-fixpoint-Lösung, die wie folgt definiert ist:

Definition 4.1.7 (MFP-Lösung)
Sei $D = (G, V, [\![\ ]\!])$ mit $G = (N, E, s, e)$ ein Datenflußproblem. Dann wird für alle $n \in N$ $\mathbf{pre}(n)$ als die Information definiert, die am Eingang eines Knotens gilt, und $\mathbf{post}(n)$ ist der Datenflußwert, der am Ausgang des Knotens gültig ist und somit auch die gesuchte Lösung:

$$\begin{aligned} \mathbf{pre}(n) &= \begin{cases} \perp & \text{falls } n = s \\ \bigsqcup\{\mathbf{post}(m) \mid (m, n) \in E\} & \text{sonst} \end{cases} \\ \mathbf{MFP}(n) = \mathbf{post}(n) &= [\![n]\!](\mathbf{pre}(n)) \end{aligned}$$

□

Das obige Gleichungssystem kann durch Einsetzen auch umformuliert werden zu:

$$\mathbf{MFP}(n) = \begin{cases} [\![s]\!](\perp) & \text{falls } n = s \\ [\![n]\!]\,(\bigsqcup\{\mathbf{MFP}(m) \mid (m, n) \in E) & \end{cases}$$

Kam hat in [KU77] bewiesen, daß die **MFP**-Lösung korrekt ist und, falls es sich um ein distributives Problem handelt, sogar gleich der **MOP**-Lösung ist.

[3] Daß diese Lösung nicht join-over-all-pathes-Lösung heißt, hat historische Gründe.

Satz 4.1.1 (Korrektheit)
Sei $D = (G, V, [\![\ ]\!])$ mit $G = (N, E, s, e)$ ein monotones Datenflußproblem. Dann ist die **MFP**-Lösung eine korrekte Approximation der **MOP**-Lösung, d.h.

$$\forall n \in N : \mathbf{MOP}(n) \sqsubseteq \mathbf{MFP}(n)$$

□

Satz 4.1.2 (Koinzidenz)
Sei $D = (G, V, [\![\ ]\!])$ mit $G = (N, E, s, e)$ ein distributives Datenflußproblem. Dann gilt

$$\forall n \in N : \mathbf{MOP}(n) = \mathbf{MFP}(n)$$

□

Bei den hier betrachteten Datenflußproblemen handelt es sich um sogenannte *Vorwärtsprobleme*, d.h. die gesuchte Information an einem Knoten wird aus den Informationen der Vorgänger im Graph berechnet. Es gibt aber auch Probleme, bei denen die Information an einem Knoten von den Informationen der Nachfolger abhängt. Ein Beispiel ist das Problem der lebendigen Variablen, wobei eine Variable lebendig heißt, wenn ihr Wert auf einem Pfad zum Endknoten nocheinmal benötigt wird. Diese Art der Datenflußprobleme werden *Rückwärtsprobleme* genannt. Ein solches Rückwärtsproblem kann angesehen werden als ein Vorwärtsproblem auf dem invertierten Kontrollflußgraphen $G^{-1} = (N, E^{-1}, e, s)$ mit $E^{-1} = \{(n_2, n_1) \mid (n_1, n_2) \in E\}$. Es ist also nicht nötig, diese getrennt zu behandeln.

Weiterhin ist es keine Einschränkung, nur *Vereinigungsprobleme* zu betrachten. Bei diesen wird die Vereinigung der Information berechnet, die sich auf allen Pfaden ergibt. Im Gegensatz dazu stehen *Schnittprobleme*, bei denen der Schnitt all dieser Informationen berechnet wird. Ein Schnittproblem ist äquivalent zu dem Vereinigungsproblem auf dem dualen Verband.

4.2 Algorithmische Lösungsverfahren

Es gibt verschiedene Ansätze zur Lösung von Datenflußproblemen. Die wichtigste Klasse, die sich im Laufe der Jahre herausgebildet hat, ist die der iterativen Algorithmen. Aber auch andere Verfahren, wie z.B. die Intervallanalyse, werden bereits in [Coc70] beschrieben. Im folgenden sollen nun einige iterative Verfahren beschrieben werden. Diese beruhen auf der Fixpunktiteration. Sie unterscheiden sich durch verschiedene Methoden zur Beschleunigung der Konvergenz.

Eingabe: Ein Flußgraph $G = (N, E, s, e)$, ein vollständiger Verband V, ein semantisches Funktional $[\![\;]\!] : N \rightarrow (V \rightarrow V)$.
Ausgabe: Eine Annotierung von G mit Datenflußinformationen, d.h. **Post**-Informationen bestehend aus Elementen von V, die die gültigen Datenflußinformationen am Ausgang jedes Knotens darstellen.

```
(* Initialisierung *)
FORALL n ∈ N DO
    MFP[n] := ⊥;
OD;
ready := FALSE;

(* Iterative Fixpunktberechnung *)
WHILE NOT ready DO
    ready := TRUE;
    FORALL n ∈ N DO
        X := [[n]](⊔{MFP[m] | (m, n) ∈ E});
        IF X ≠ MFP[n] THEN
            MFP[n] := X;
            ready:= FALSE;
        END;
    OD;
OD;
```

Abbildung 4.1: Intuitive Methode zur Berechnung der **MFP**-Lösung

4.2.1 Intuitive Methode

Wie bereits erwähnt, ist die **MFP**-Strategie praktisch relevant, da sie direkt eine iterative Prozedur zur Berechnung der **MFP**-Lösung spezifiziert, die in Abbildung 4.1 dargestellt ist. Dieser Algorithmus terminiert, wie in Abschnitt 3.2 erläutert, wenn das Datenflußproblem monoton ist und der zugrunde liegende Verband V die aufsteigende Kettenbedingung erfüllt. Seine Komplexität wird bestimmt von der Anzahl K der Iterationen, die bis zur Stabilisierung benötigt werden, den Kosten zum Vergleich zweier Verbandselemente und der Komplexität der Transferfunktionen.

4.2.2 Worklist Iteration

Man kann die Konvergenz des Verfahrens beschleunigen, indem man nicht in

Eingabe: Ein Flußgraph $G = (N, E, s, e)$, ein vollständiger Verband V, ein semantisches Funktional $[\![\ ]\!] : N \to (V \to V)$.
Ausgabe: Eine Annotierung von G mit Datenflußinformationen, d.h. **Post**-Informationen bestehend aus Elementen von V, die die gültigen Datenflußinformationen am Ausgang jedes Knotens darstellen.
Bemerkung: Die Variable workset kontrolliert den iterativen Prozeß. Ihre Elemente sind Knoten $n \in N$ von G

(* Initialisierung *)
FORALL $n \in N$ **DO**
  MFP$[n] := [\![n]\!](\perp)$;
OD;
workset := $\{s\}$;

(* Iterative Fixpunktberechnung *)
WHILE workset $\neq \emptyset$ **DO**
  LET $n \in$ workset
    BEGIN
      workset := workset $\setminus\{n\}$;
      $X := [\![n]\!]\,(\bigsqcup\{$**MFP**$[m] \mid (m, n) \in E\})$;
      IF $X \neq$ **MFP**$[n]$ **THEN**
        MFP$[n] := X$;
        workset := workset $\cup\{m \mid (n, m) \in E\}$;
      FI
    END
OD.

Abbildung 4.2: Die Berechnung der **MFP**-Lösung mittels Worklist-Verfahren

jeder Iteration die Werte an sämtlichen Knoten neu berechnet, sondern nur für diejenigen, an denen sich die Eingangsinformation verändert hat. Um sich diese Knoten zu merken, wird eine Worklist eingeführt. Der resultierende Algorithmus ist in Abbildung 4.2 zu sehen.

Weitere Beschleunigungen lassen sich dadurch erreichen, daß man die Auswahl der Elemente aus der Worklist durch bestimmte Heuristiken vornimmt. Besonders bewährt hat sich dabei die Sortierung der Knoten durch folgendes Verfahren:

1. Berechne die starken Zusammenhangskomponenten SZK ([Meh84]) des Kontrollflußgraphen G.

Eingabe: Ein Flußgraph $G = (N, E, s, e)$, ein vollständiger Verband V, ein semantisches Funktional $[\![\ ]\!] : N \rightarrow (V \rightarrow V)$.
Ausgabe: Eine Annotierung von G mit Datenflußinformationen, d.h. **Post**-Informationen bestehend aus Elementen von V, die die gültigen Datenflußinformationen am Ausgang jedes Knotens darstellen.
Bemerkung: Die Variable Workset kontrolliert den iterativen Prozeß. Ihre Elemente sind Knoten $n \in N$ von G

(* Initialisierung *)
FORALL $n \in N$ **DO**
  MFP$[n] := \perp$;
OD;
ready := **FALSE**;

(* Iterative Fixpunktberechnung *)
FOR $i := 0$ **TO** $L(V)$ **DO**
  FORALL $n \in N$ **DO**
    $X := [\![n]\!] (\bigsqcup\{\ \mathbf{MFP}[m] \mid (m, n) \in E\})$;
  OD;
OD;

Abbildung 4.3: Der Algorithmus zur Berechnung des Fixpunktes nach Nielson

2. Sortiere die SZK des resultierenden Graphen nach ihren DFS-Nummern[4].

3. Sortiere die Knoten der einzelnen SZK ebenfalls durch DFS.

4.2.3 Bounded Fixpoint Iteration

Das Hauptproblem der Fixpunkt Iteration ist die Zeit- und Platzkomplexität. Besonders dann, wenn der zugrunde liegende Verband sehr komplex ist, können die Kosten, zwei Iteranden zu vergleichen sehr hoch sein. Dies ist z.B. der Fall, wenn Graphen auf strukturelle Gleichheit untersucht werden sollen. Da dieser Test in einer Worklist Iteration nach jeder Neuberechnung der Information an einem Knoten vorgenomen werden muß, um zu überprüfen, ob der Fixpunkt bereits erreicht wurde, geht seine Komplexität stark in die Komplexität des Algorithmus ein. Der Vorschlag, der in [NN92] gemacht wird, um dieses Problem zu umgehen, besteht

[4] die z.B. in [Hec77] beschriebene depth-first-search Numerierung

Programm	obere Schranke $L(V)^{\|N\|}$ in Tausend	tatsächliche Iterationen
`cdecl`	19584	55
`ed`	11399	44
`fft`	936	18
`flex`	154829	139
`flops`	218	30
`gzip`	72192	146
`heap`	154	8
`linpack`	1065	16
`ratfor`	27976	102
`twig`	24655	36
`xmodem`	29383	99

Es wurde an verschiedenen Programmen (vergleiche mit Anhang B) eine vollständige Konstantenanalyse durchgeführt. Der zugrunde liegende Verband ist dabei ein n-Tupel aus dem flachen Verband der möglichen Werte, wobei n die Anzahl der Variablen ist. Die Anzahl der tatsächlichen Iterationen wurde bei einer Worklist Iteration mit zufälliger Auswahl gemessen. Dabei wurde die universelle Methode aus Kapitel 5.2.4 zur interprozeduralen Analyse benutzt.

Abbildung 4.4: Vergleich der theoretischen Obergrenze der maximal nötigen Iterationen und der tatsächlich benötigten Anzahl

$$\begin{aligned}
L\left(flat(V)\right) &= 2 \\
L(V_1 \times \cdots \times V_n) &= L(V_1) + \cdots + L(V_n) \\
L(V_1 +\!+ \cdots +\!+ V_n) &= max\left(L(V_1), \cdots, L(V_n)\right) \\
L(M \to V) &= L(D)^{|M|} \\
L\left(2^M\right) &= |M|
\end{aligned}$$

Abbildung 4.5: Die Berechnung der längsten Ketten von Verbänden

darin, eine Anzahl k von Iterationen zu bestimmen, bei der garantiert ist, daß der Fixpunkt erreicht ist, um so die Vergleiche für den Stabilitätstest zu sparen.

Die Idee dabei ist, daß für einen Verband mit endlichen Ketten die Länge L der längsten Kette durch strukturelle Induktion über die Konstruktion des Verbandes folgendermaßen bestimmt werden kann: Falls V explizit angegeben ist, kann L abgezählt werden. Ansonsten gelten die in Abbildung 4.5 gezeigten Regeln.

Da der Fixpunkt auf allen Informationen gleichzeitig berechnet wird, handelt es sich streng genommen um eine Fixpunktiteration auf einem n-Tupel von Elementen aus dem zugrunde liegenden Verband V, falls der zugehörige Kontrollflußgraph n Knoten hat. Daher kann die gesuchte Zahl L für ein Datenflußproblem $D = (G, V, F, [\![\ ]\!])$ mit $G = (N, E, s, e)$ mit Hilfe des obigen Gleichungssystems berechnet werden als $L = (L(V))^{|N|}$.

Dieses Ergebnis ist nur eine obere Schranke für die maximal benötigte Anzahl von Iterationen. Je nach dem eingesetzten Verfahren zur Fixpunktberechnung kann L weiter verfeinert werden. Falls z.B. die oben beschriebene Iterationsreihenfolge mittels SZK und DFS verwendet wird, ist L gleich der Loop Connectedness von G plus eins multipliziert mit $L(V)$. Dabei ist die Loop Connectedness nach [Hec77] definiert als:

Definition 4.2.1 (Loop Connectedness)
Sei $G = (N, E, s, e)$ ein Flußgraph und sei T ein aufspannender Baum[5] von G. Man definiert $d(G, T)$, die *Loop Connectedness*[6] von G bezüglich T als die größte Anzahl von Rückwärtskanten in einem zyklenfreien Pfad von G. (Falls G reduzierbar ist, wird die Definition unabhängig von dem gewählten T). □

Also kann die Loop Connectedness abgeschätzt werden durch die gesamte Anzahl von Rückwärtskanten in einem aufspannenden Baum. Diese kann zur selben Zeit berechnet werden, wie die benötigte Depth First Order.

Der resultierende Algorithmus ist in Abbildung 4.3 dargestellt. Allerdings ist zu bemerken, daß Einsparungen mit diesem Verfahren nur gemacht werden können, wenn der benutzte Verband tatsächlich extrem kompliziert ist, da die abgeschätzte Obergrenze L meist wesentlich größer ist als die tatsächlich benötigte Anzahl der Iterationen zur Stabilisierung. Vergleiche dazu auch Tabelle 4.4.

4.2.4 Widening and Narrowing

Eine andere Möglichkeit, um die Terminierung der Datenflußanalyse zu erzwingen, besteht darin, die in [CC77, CC92a] beschriebene **Widening and Narrowing** Technik zu verwenden. Sie erreicht eine Terminierung sogar, wenn der zugrunde liegende Verband nicht die endliche Kettenbedingung erfüllt. Aber auch in anderen Verbänden kann damit die Iteration zugunsten eines Informationsverlustes beschleunigt werden.

[5] Depth First Spanning Tree, d.h. der Baum, der die Wege durch einen Graphen bei einer Durchmusterung mit Tiefensuche beschreibt.

[6] Wird in [ASU86] auch Tiefe genannt.

Dies wird erreicht, indem man die bei der Iteration auftretenden Datenflußwerte durch größere Verbandselemente abschätzt, welche immer eine korrekte Approximation sind. Eine Kette von solchen Abschätzungen muß so gewählt werden, daß sie endlich ist und somit der Prozeß terminiert. Anschließend kann man die gefundene Abschätzung noch verbessern, indem man sich dem Fixpunkt von oben durch eine weitere Iteration nähert.

Es werden also vor der Iteration die Datenflußinformationen aller Knoten mit der minimalen Information initialisiert. Dann wird mittels des Widening eine obere Annäherung an den Fixpunkt berechnet. Anschließend kann mit dem Narrowing Operator die Annäherung weiter verbessert werden.

Widening

Ein Widening Operator $\bigtriangledown \in [V \times V \to V]$ wird so gewählt, daß

$$\begin{array}{c} \forall x, y \in V : x \sqsubseteq x \bigtriangledown y \\ \text{und} \\ \forall x, y \in V : y \sqsubseteq x \bigtriangledown y \end{array}$$

ist. Außerdem soll für alle aufsteigenden Ketten $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n$ die Kette $y_0, y_1, \cdots, y_n$ definiert durch

$$\begin{array}{rcl} y_0 & = & x_0 \\ & \vdots & \\ y^{i+1} & = & y^i \bigtriangledown x^i \end{array}$$

nicht streng aufsteigend sein. Die aufwärts gerichtete Iterationsfolge mit Widening ist definiert als:

$$\begin{array}{rcll} X^0 & = & \bot & \\ X^{i+1} & = & X^i & \text{, falls } F(X^i) \sqsubseteq X^i \\ & = & X^i \bigtriangledown F(X^i) & \text{, sonst} \end{array}$$

Für vollständige Verbände, welche die aufsteigende Kettenbedingung erfüllen, ist die Operation ‚Bildung der kleinsten oberen Schranke' ein Widening.
Der Grenzwert der Iteration ist ein Präfixpunkt von F, der nach Satz 3.2.1 größer oder gleich dem kleinsten Fixpunkt ist.

Narrowing

Die Annährung an den Fixpunkt kann durch Narrowing weiter verbessert werden. Für einen Narrowing Operator $\bigtriangleup \in [V \times V \to V]$ muß gelten:

$$\forall x, y \in V : (y \sqsubseteq x) \Rightarrow (y \sqsubseteq (x \bigtriangleup y) \sqsubseteq x)$$

Außerdem soll für alle absteigenden Ketten $x_0 \sqsupseteq x_1 \sqsupseteq x_2 \cdots \sqsupseteq x_n$ die Kette der $y_1, y_2, \cdots, y_n$ mit

$$\begin{aligned} y_0 &= x_0 \\ &\vdots \\ y^{i+1} &= y^i \bigtriangleup x^{i+1} \end{aligned}$$

nicht streng absteigend sein. Die absteigende Iterationsfolge mit Narrowing ist definiert als:

$$\begin{aligned} X^0 &= \text{Grenze der Aufwärtsiteration} \\ X^{i+1} &= X^i \bigtriangleup F(X^i) \end{aligned}$$

Für vollständige Verbände, welche die Kettenbedingung erfüllen, ist die Bildung der größten unteren Schranke ein Narrowing.

Kapitel 5

Interprozedurale Datenflußanalyse

Bisher wurden nur Verfahren zur intraprozeduralen Analyse aufgezeigt. Prozeduren werden aber nicht nur in imperativen Sprachen wie Pascal intensiv genutzt, sondern spielen auch eine wesentliche Rolle in logischen Sprachen wie z.B. Prolog die sich als Konjunktionen von Prozeduraufrufen ansehen lassen. So ist es von großer Wichtigkeit, Programme in ihrer Gesamtheit zu analysieren.

Bei der Lösung von Datenflußproblemen im Kontext der interprozeduralen Analyse besteht die größte Schwierigkeit in der Behandlung von rekursiven Prozeduren. Ein weiteres Problem stellt der modulare Aufbau von Programmen dar, der zu der weitgehend ungelösten Schwierigkeit führt, daß nicht alle Aufrufstellen einer zu analysierenden Prozedur bekannt sind bzw. daß in der zu analysierenden Prozedur ein Aufruf einer Funktion erfolgt, deren Definition nicht bekannt ist. Auf die drei am meisten etablierten Verfahren soll nun genauer eingegangen werden.

5.1 Lösungsansätze

5.1.1 Triviale Methode

Das einfachste Verfahren besteht darin, Prozeduraufrufe als einfache Anweisungen anzusehen und bei deren Auftreten alle verfügbaren Informationen zu zerstören. Dieses Verfahren bringt, wie leicht einzusehen ist, keine gute Information bei Prozeduraufrufen, dafür ist es unkompliziert auszuführen, falls man bereits ein intraprozedurales Verfahren implementiert hat.

Man kann zur Verbesserung dieser Methode auch nur die Information an einer Aufrufstelle zerstören, die auf Objekten beruht, welche von der aufgerufenen Prozedur verändert werden können. Dazu benötigt man eine vorangegangene Analysephase, die für jede Prozedur die Menge der potentiell veränderten Variablen berechnet. Dies ist wiederum ein Datenflußproblem.

5.1.2 Inlining

Ein exakteres Vorgehen ist das Prozedurinlining. Es besteht darin, den Rumpf einer Prozedur textuell an die Stelle ihres Aufrufes zu kopieren und das resultierende Programm intraprozedural zu analysieren. Dieses Verfahren bietet den Vorteil, daß es ohne wesentlichen Mehraufwand durchzuführen ist, daß verschiedene Aufrufe einer Prozedur getrennt analysiert werden und es so zu keiner Vermischung (Bildung der oberen Schranke) der Kontexte verschiedener Aufrufstellen, kommen kann. Da auf diese Art und Weise nur nicht rekursive Funktionen behandelt werden können, ist das Verfahren in der Praxis wenig nützlich. Außerdem kann es auch im Falle nicht rekursiver Programme vorkommen, daß diese durch Inlining exponentiell wachsen.

5.1.3 Effektberechnung

In diesem Verfahren wird von jeder Prozedur eine abstrakte Version berechnet, die die Abhängigkeit der Datenflußinformation am Ausgang der Prozedur von der am Eingang der Prozedur gültigen Information beschreibt. Eine solche Funktion wird in der Literatur (siehe [GT93]) häufig *Effektfunktion* oder *Jump-Funktion* genannt.

Diese Funktion kann nun als Transferfunktion für die Aufrufstellen der zugehörigen Prozedur benutzt werden. D.h. es kann eine normale intraprozedurale Iteration auf dem Kontrollflußgraphen durchgeführt werden. Da die Prozeduren eines Programms simultan rekursiv sein können, muß die Berechnung der Effektfunktionen aller Prozeduren gleichzeitig erfolgen.

Dieses Verfahren beinhaltet mehrere Schwierigkeiten: Zum einen muß die Analyse nun zweistufig erfolgen, wobei sich die Algorithmen für die zwei Phasen wesentlich voneinander unterscheiden, was die interprozedurale im Vergleich zu einer intraprozeduralen Analyse erheblich verkompliziert. Zum anderen muß eine Darstellung für die Effektfunktionen gefunden werden, wozu oftmals eine extensionale Beschreibung verwendet wird. Das schränkt allerdings die Menge der benutzbaren Verbände stark ein, da diese dann nur noch endlich sein dürfen, um weiterhin die Terminierung der Analyse zu garantieren. Aber auch wenn diese Bedingung erfüllt ist, kann die Repräsentierung der Funktionen exponentiell wachsen, wie in [CCKT86] beschrieben.

Dieses Verfahren kann trotzdem besonders dann sehr nützlich sein, wenn der zugrunde liegende Verband endlich ist, wie es in vielen eingeschränkten Versionen der Konstantenpropagation der Fall ist. Dann hat diese Methode den Vorteil, präzise

Ergebnisse zu liefern, wie in [RHS95, SRH95] beschrieben.

Trotz aller Schwierigkeiten dieses Verfahrens ist es möglich, dieses in PAG einzubauen. Es läßt sich dann, wegen seiner theoretischen Beschränkungen, nur für endliche Verbände verwenden. Dabei kann ein Algorithmus benutzt werden, der Funktionen nicht direkt manipuliert, d.h. also nicht die Vereinigung oder die Komposition ganzer Funktionen bildet, sondern sich auf Funktionsanwendungen und die Vereinigung auf Verbandselementen beschränkt. Der allgemeine Algorithmus kann mit den verschiedenen Iterationsformen aus Kapitel 4.2 kombiniert werden. Er ist, gepaart mit einer Worklist Iteration, in Abbildung 5.1 zu sehen.

5.1.4 Call String Ansatz

Diese Methode wurde von Sharir und Pnueli in [SP81] vorgestellt. Bei ihr wird die interprozedurale Analyse mit der intraprozeduralen Phase gemischt. Dazu wird aus dem gesamten Programm ein einziger zusammenhängender Kontrollflußgraph erstellt, indem man Prozeduraufrufe und deren Rückkehr als einfache Kontrollübergangsinstruktionen betrachtet. Um dabei die Datenflußinformation nur entlang solcher Pfade zu propagieren, die auch tatsächlich potentielle Ausführungspfade darstellen, wird die Datenflußinformation mit einer kodierten Form ihrer Aufrufgeschichte versehen. Dieser Ansatz hat den Vorteil, daß er sich leicht in einen approximativen Algorithmus verwandeln läßt, der einige Details des interprozeduralen Flusses verliert, aber effizient zu implementieren ist.

Ein **Call String** γ ist ein Tupel von Prozeduraufrufen $c_1, c_2, \cdots, c_j$, das eine Aufrufgeschichte darstellt. Dazu muß es einen Pfad durch den Kontrollflußgraphen geben, auf dem eine solche Abfolge von Aufrufen möglich ist. Die Menge der möglichen Call Strings Γ ist also vom Programm abhängig und unendlich, falls das Programm (simultan) rekursive Prozeduren enthält.

Ein gegebener Datenflußrahmen (V, F) wird zunächst in einen Rahmen (V^*, F^*) transformiert. Dabei ist $V^* = \Gamma \rightarrow V$ die Menge aller Funktionen, die Call Strings Datenflußelemente zuordnet. Dieser Verband erfüllt aber im allgemeinen nicht die endliche Kettenbedingung, es sei denn, Γ ist endlich. Daher terminiert die Analyse nicht zwangsläufig.

Eine Transferfunktion f_n wird erweitert zu einer Funktion f_n^* auf V^*, indem man sie für alle Call Strings γ auf den Flußwert anwendet. Falls n allerdings ein Aufruf oder der erste Knoten nach der Rückkehr aus einer Prozedur ist, wird der berechnete Wert entsprechend zu einem anderen Call String verschoben. F^* ergibt sich dann als der übliche Abschluß dieser erweiterten Transferfunktionen.

Eingabe: Eine Sammlung von Flußgraphen $G^* = \bigcup_{P_i} G_{P_i}$ für alle Prozeduren P_i eines Programms. Ein vollständiger Verband V, ein semantisches Funktional $[\![\,]\!] : N^* \to (V \to V)$.

Bemerkung: Die Effektfunktionen Φ werden repräsentiert durch ein zweidimensionales partiell definiertes Array PHI: $N^* \times V \to V$, so daß für jedes $n \in N^*, x \in V$, $\mathrm{PHI}(n,x)$ gleich $\Phi_{(\mathrm{ENTRY}_P,n)}(x)$ ist, wobei P die Prozedur ist, die n enthält. Die Variable workset kontrolliert den iterativen Prozeß. Ihre Elemente sind Paare aus Knoten $n \in N^*$ und Datenflußwerten $x \in V$, für die sich PHI(n,x) verändert hat und deren neuer Wert bisher nicht zu den Nachfolgern von n propagiert wurde.

```
(* Initialisierung *)
PROCEDURE init;
   FORALL n ∈ N*, x ∈ V DO
      PHI(n, v) := ⊥;
   OD;
   workset := {(ENTRY_main, ⊥)};
END

(* Propagierungsfunktion *)
PROCEDURE propagate(x ∈ V, y ∈ V, n ∈ N*)
   z := PHI(n, x) ⊔ y;
   IF z ≠ PHI(n, x) THEN
      PHI(n, x) := z;
      workset := workset ∪ {(n, x)};
   FI
END

(* Berechnung der post Informationen *)
PROCEDURE post;
   FORALL n ∈ N* DO
      post(n) = [[n]] (⊔ { PHI(n, x) | x ∈ V });
   OD
END
```

Abbildung 5.1: Der Algorithmus zur Berechnung der interprozeduralen Lösung eines Datenflußproblems mit endlichem Verband mittels der Berechnung von Effektfunktionen. Die Fortsetzung ist in Abbildung 5.2 zu sehen.

```
PROCEDURE jump_functions; (* Iterative Berechnung der Funktionen PHI *)
  WHILE workset ≠ ∅ DO
    LET
      (n, x) ∈ workset;
      y = PHI(n, x);
      IN
        workset := workset \{(n,x)};
        IF n ≃ CALL P in q THEN
          z := PHI(EXIT_P, y);
          IF z ≠ ⊥ THEN
            m := RETURN Knoten zu n;
            propagate(x, z, m);
          ELSE
            propagate(y, y, ENTRY_P);
          FI
        ELIF n ≃ EXIT P THEN
          FORALL c = CALL P DO
            m := RETURN Knoten zu c;
            FORALL u ∈ V WHERE PHI(c, u) = x DO
              propagate(u, y, m);
            OD
          OD
        ELSE
          FORALL m Nachfolger von n DO
            propagate(x,[[m]](y),m);
          OD
        FI
      END
  OD
END

(* Hauptprogramm *)
  init();
  jump_functions();
  post();
END.
```

Abbildung 5.2: Fortsetzung von Abbildung 5.1

Falls V endlich ist, ist es ausreichend, Call Strings der Länge höchstens $K * |V|^2$ zu betrachten. Dabei ist K die Anzahl der Prozeduraufrufe im betrachteten Programm und $|V|$ die Mächtigkeit des Verbandes V. Dann ist die Menge der zu betrachtenden Call Strings endlich, und so kann die Terminierung garantiert werden. Leider wird diese Zahl in der Praxis aber sehr groß, so daß auch dann kein effizienter Algorithmus zu erwarten ist.

5.2 Gewählter Ansatz

5.2.1 Intuition

Für den hier gewählten Ansatz waren die folgenden Punkte wichtig:

- Jede Analyse, die intraprozedural terminiert, muß auch mit dem gewählten interprozeduralen Verfahren terminieren. Dieser Punkt schließt bereits die Verwendung der Effektberechnung aus.
- Der zusätzliche Aufwand für die interprozedurale Analyse sollte so gering wie möglich sein. Das gilt vor allem für den nicht konstanten Anteil der Analysen, der spezifiziert werden muß (siehe Kapitel 6).
- Die interprozedurale Analyse sollte in ihrer Platz– und Zeitkomplexität so effizient bleiben, daß sie praktisch durchführbar ist. Am besten wäre es, wenn sie durch zusätzliche Laufzeitparameter veränderbar wäre.
- Die Analyseergebnisse sollen in dem gesteckten Rahmen so präzise wie möglich sein.

Leider ist es unmöglich (Beweis in [SP81]), für alle intraprozedural terminierenden Algorithmen eine präzise interprozedurale Lösung zu erhalten. Das geht nur, wenn der zugrunde liegende Verband endlich oder distributiv ist, oder falls die zu analysierenden Programme nicht rekursiv sind. Alle diese Einschränkungen sind für einen universellen Analysator Generator wie PAG nicht annehmbar. Daher wurde darauf verzichtet, präzise Lösungen für Datenflußprobleme zu suchen.

Der hier benutzte Ansatz ist mit der Call String Methode eng verwandt, benutzt aber eine andere Art der Implementierung. Dieser Weg wurde gewählt, da man damit leicht Zeit gegen Platz bei der Berechnung eintauschen kann, indem man die Länge der Call Strings verändert. Auf diesen Punkt wird in Kapitel 5.2.4 genauer eingegangen. Man kann feststellen, daß in den meisten tatsächlich benutzten Programmen Aufrufe einer Prozedur von verschiedenen Stellen im Programmtext mit verschiedenen Werten für globale Variablen und Parameter der Prozedur gemacht werden. Zusätzlich ist eine große Anzahl von Prozeduraufrufen nicht rekursiv. Es liegt also nahe, verschiedene Aufrufe einer Prozedur, die von unterschiedlichen Stellen im Quelltext aus erfolgen, getrennt zu analysieren, wie es beim Inlining der Fall ist. Allerdings sind solche Aufrufe zwar nicht rekursiv aber oftmals in mehreren Stufen verschachtelt, so daß es sich auch lohnen kann, Pfade durch den Aufrufbaum einer gewissen Länge getrennt zu betrachten. Dies soll durch das folgende Beispiel geklärt werden.

Beispiel 5.2.1
Es sei folgendes C Programm mittels einer Konstantenpropagation zu analysieren:

```
static int i;
void main(void) {
   i = 1;
   r();
   print(i);
   r();
   print(i);
}

void r(void) {
   q();
}

void q(void) {
   i = i + 1;
}
```

In Bild 5.3 ist der *Aufrufgraph* des Programms zu sehen. Das ist ein Graph, in dem die Prozeduren Knoten sind, und eine Kante von P_1 nach P_2 einem möglichen Aufruf der Prozedur P_2 in P_1 darstellt.

Die Genauigkeit der Analyse hängt davon ab, für welche Zahl n man alle Aufrufketten der Länge kleiner gleich n getrennt analysiert, d.h. keine Datenflußinformation zusammenmischt, falls der Weg im Aufrufgraphen, auf dem die aktuelle Inkarnation der Prozedur vom Hauptprogramm aus erreicht wurde, maximal n Kanten enthält.

Wählt man für dieses Programm $n = 2$, so kann man herausfinden, daß der Aufruf von `q` aus `r` heraus zweimal mit verschiedenen, aber für den jeweiligen Aufruf konstanten Werten ausgeführt wird. Falls man aber nur Aufrufketten der Länge

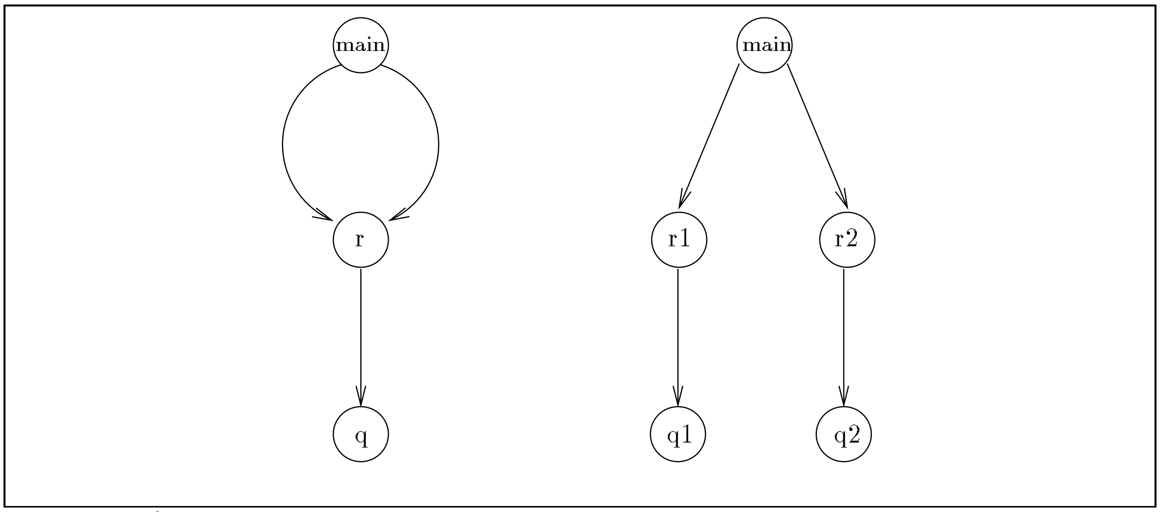

Abbildung 5.3: Die Aufrufgraphen zu Beispiel 5.2.1 und Beispiel 5.2.2

eins trennt und nachfolgende Aufrufe zusammenmischt, kann nur herausgefunden werden, daß `i` beim Aufruf von `q` einen unbekannten Wert hat. □

Um die Trennung der verschiedenen Aufrufketten zu erreichen, wird nicht, wie in der Call String Methode, die Aufrufgeschichte mit in den Verband aufgenommen, sondern es wird ein Feld von Datenflußelementen pro Kontrollflußknoten eingeführt, anstatt nur ein Element pro Knoten zu benutzen. Die einzelnen Elemente dieses Vektors entsprechen (statisch) festgelegten Aufrufpfaden. Um auch rekursive Funktionen behandeln zu können, werden die Daten von Aufrufpfaden ab einer bestimmten Länge mit einer join Operation zusammengemischt.

Mittels der Länge der längsten getrennt zu analysierenden Aufrufkette, die vor der Analyse festgelegt wird, kann auch die bereits erwähnte Feinabstimmung zwischen Dauer und Platzbedarf der Analyse und der erreichten Genauigkeit erfolgen.

Nachdem die Analyse beendet ist, kann der Datenflußvektor auf verschiedene Arten genutzt werden. Zum einen kann die Vereinigung aller Elemente zur Optimierung des Prozedurknotens verwendet werden, da er eine korrekte Approximation für alle Aufrufpfade, die zu dieser Prozedur führen, darstellt. Andererseits können die verschiedenen Datenflußelemente einzeln oder zu Gruppen zusammengefaßt benutzt werden, da sie korrekte Werte für einzelne Aufrufpfade darstellen. So können zwei Kopien der Prozedur erzeugt werden, die anschließend einzeln optimiert werden, d.h. daß die Prozedur spezialisiert werden kann, und mittels der Datenflußvektoren kann geprüft werden, für welche Aufrufpfade sich eine solche Spezialisierung lohnt.

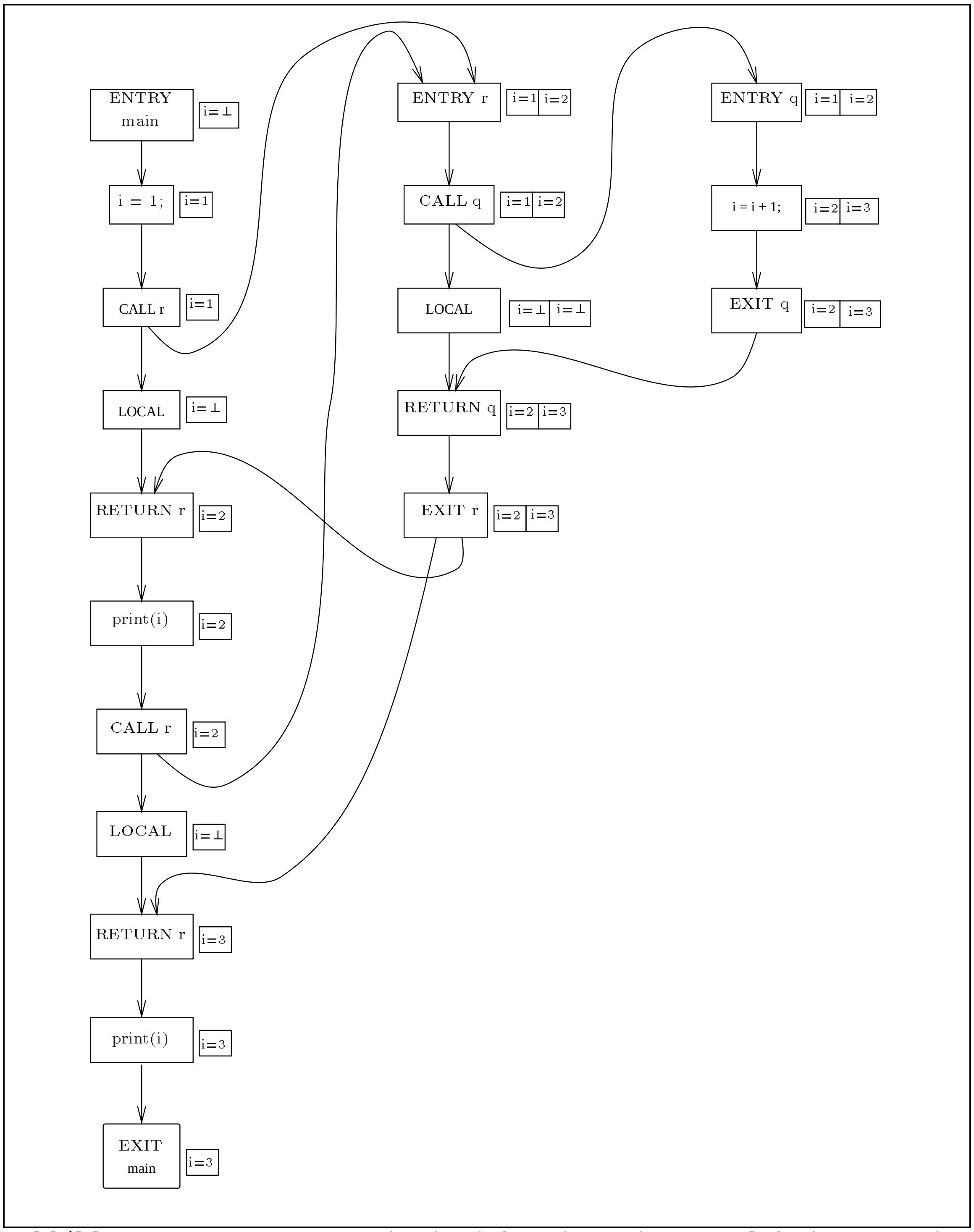

Abbildung 5.4: Der intraprozedurale Flußgraph mit der Datenflußinformation der Konstantenpropagation zu Beispiel 5.2.1

Beispiel 5.2.2
In Bild 5.3 ist der Datenflußgraph des Programms aus Beispiel 5.2.1 gezeigt. Der genaue Aufbau des Graphen wird noch erläutert. Hier genügt es zu wissen, daß neben den Knoten der Vektor der berechneten Datenflußinformation steht. Man kann am Prozedurkopf von `q` (ENTRY q) erkennen, daß `q` von zwei verschiedenen Stellen aufgerufen wird. Dabei ist einmal i gleich 1 und das andere mal i gleich 2. Da i in q verwendet wird, soll `q` spezialisiert werden. Das Ergebnis ist das folgende Programm, in dem weiter optimiert werden kann:

```
static int i;
void main(void) {
   i = 1;
   r1();
   print(2);
   r2();
   print(3);
}

void r1(void) {
   q1();
}

void r2(void) {
   q2();
}

void q1(void) {
   i = 2;
}

void q2(void) {
   i = 3;
}
```

□

5.2.2 Formale Beschreibung

Nun soll die oben intuitiv beschriebene Methode formalisiert werden.

Definition 5.2.1 (Supergraph)
Ein Programm P, das aus den Prozeduren $P_0, \cdots, P_n$ besteht, wobei P_0 die Hauptprozedur ist, wird repräsentiert, indem man einen gerichteten Graphen $G^* = (N^*,$

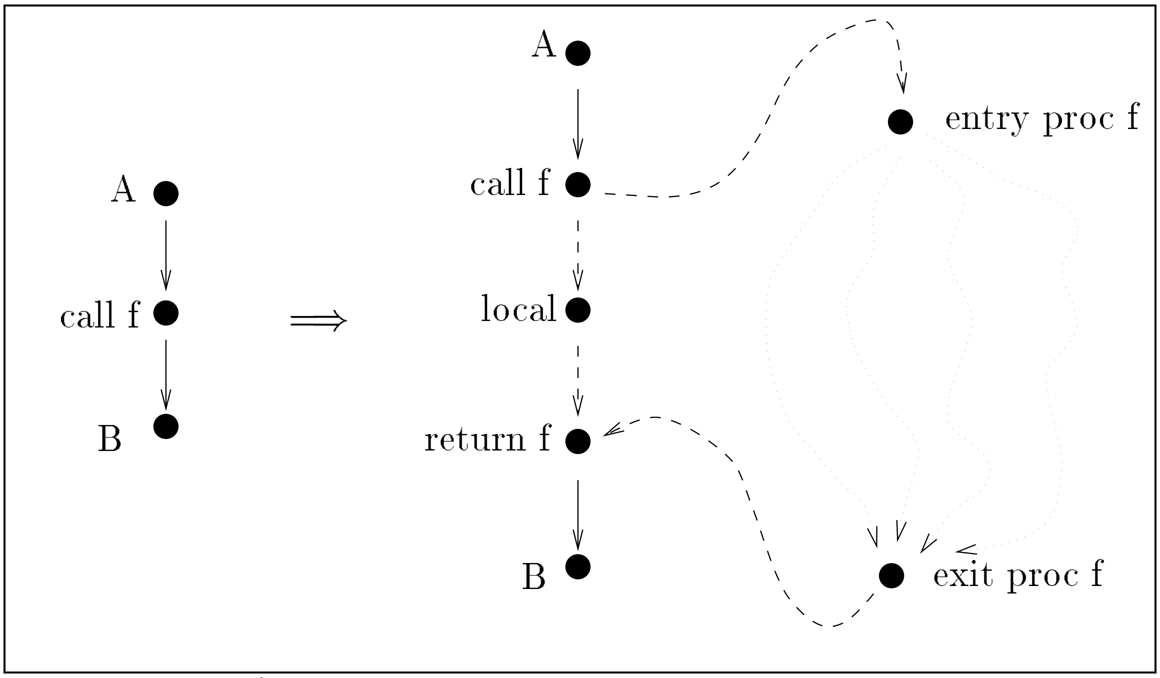

Abbildung 5.5: Die Transformation eines Call-Knotens

E^*, s^*, e^*), *Supergraph* genannt, verwendet. G^* besteht aus einer Anzahl von Flußgraphen $G_0, \cdots, G_n$, die jeweils die Prozeduren $P_0, \cdots, P_n$ repräsentieren. Jeder Flußgraph $G_i = (N_i, E_i, s_i, e_i)$ ist ein intraprozeduraler Kontrollflußgraph für die Prozedur P_i nach Definition 4.1.1, abgesehen davon, daß ein Prozeduraufruf durch drei Knoten repräsentiert wird, von einem *Call*-Knoten, einem *Local*– und einem *Return*-Knoten.

Zusätzlich zu den intraprozeduralen Kanten, die die Knoten der einzelnen Flußgraphen miteinander verbinden, hat G^* für jeden Prozeduraufruf einer Prozedur P, der durch den Aufrufknoten c, den Local-Knoten l und den Return-Knoten r repräsentiert wird, folgende Kanten:

- eine Aufrufkante von c zum Startknoten von P,
- eine Kante von c nach l,
- eine Rückkehrkante vom Endknoten von P nach r und
- eine Kante von l nach r. □

Die Transformation von Call-Knoten wird in Abbildung 5.5 gezeigt. Der Local-Knoten ermöglicht es, Datenflußinformationen, die von einem Prozeduraufruf nicht geändert werden können, vom Call– zum Return-Knoten zu propagieren. Dies können z.B. lokale Variablen sein, die von der aufgerufenen Prozedur nicht (etwa durch Aliase) verändert werden können. Der Local-Knoten wurde gewählt (statt eine Kante direkt vom Call– zum Return-Knoten zu ziehen), um die Möglichkeit zu haben, die propagierte Information mittels einer Transferfunktion zu beeinflussen. Man kann die Transferfunktion des Local-Knotens so wählen, daß sie konstant $\perp$ zurückliefert. Dann wird die implizit vor dem Return-Knoten ausgeführte Vereinigungsoperation immer den Wert liefern, den die Prozedur zurückliefert. Man kann aber, je nach Semantik der zu analysierenden Sprache und der durchgeführten Analyse, andere Transferfunktionen für den Return– und den Local-Knoten wählen und so eine bessere Modellierung der Semantik der Sprache erreichen. Dies soll im folgenden Beispiel erläutert werden.

Beispiel 5.2.3
Um in einer Konstantenpropagation das Verhalten lokaler Variablen in C zu modellieren, setzt man an den Local-Knoten all diejenigen Variablen auf $\perp$, die von der aufgerufenen Prozedur modifiziert werden können. Zusätzlich werden dann an dem Exit-Knoten der aufgerufenen Prozedur alle übrigen Variablen auf $\perp$ gesetzt. So kommen am Return-Knoten die lokalen Variablen der Prozedur über den Local-Knoten und alle übrigen über den Return-Knoten der Prozedur an.

Dieses Vorgehen erfordert in Sprachen, die es ermöglichen, Zeiger zu übergeben, eine vorherige Aliasanalyse, um diejenigen Variablen zu finden, die von einer Prozedur verändert werden können. In einer Sprache, die kein Zeigerkonzept und keine Referenzparameter besitzt, sind dies nur die Variablen, die vom Rumpf der Prozedur aus sichtbar sind. □

So ermöglicht es der Local-Knoten, flexibel auf die Erfordernisse verschiedener Analysen und Quellsprachen einzugehen.

Ein Beispiel für ein System von Kontrollflußgraphen und den zugehörigen Supergraphen ist in den Abbildungen 5.6 und 5.7 zu sehen.

Bei der Konstruktion eines Supergraphen kann folgendes Problem auftreten: Falls der Name der aufgerufenen Prozedur nicht im Quelltext steht, sondern in einer Prozedurvariablen enthalten ist, ist statisch nicht zu ermitteln, welche Prozedur aufgerufen wird. Daher müssen Kanten zu allen potentiell aufrufbaren Prozeduren gezogen werden. Falls sehr viele Aufrufe von Prozedurvariablen gemacht werden, kann der resultierende Flußgraph sehr dicht werden.

In einem Supergraphen kann eine normale Datenflußanalyse durchgeführt werden, falls man ihn als intraprozeduralen Kontrollflußgraphen betrachtet. Diese Analyse erfordert, im Gegensatz zu der Effektberechnung, keinen zusätzlichen Aufwand außer demjenigen, den nötigen Supergraphen zu berechnen.

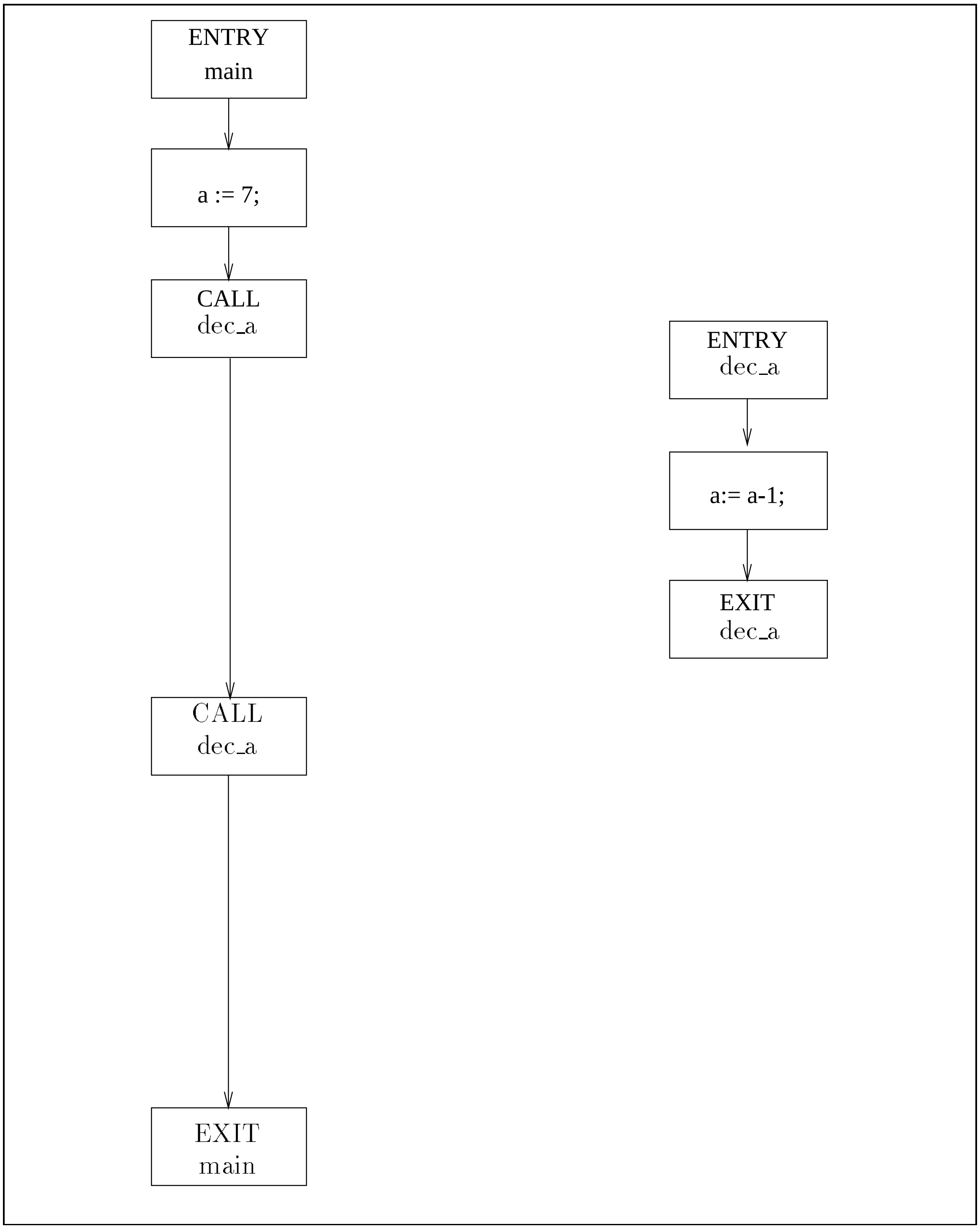

Abbildung 5.6: Ein System von Kontrollflußgraphen

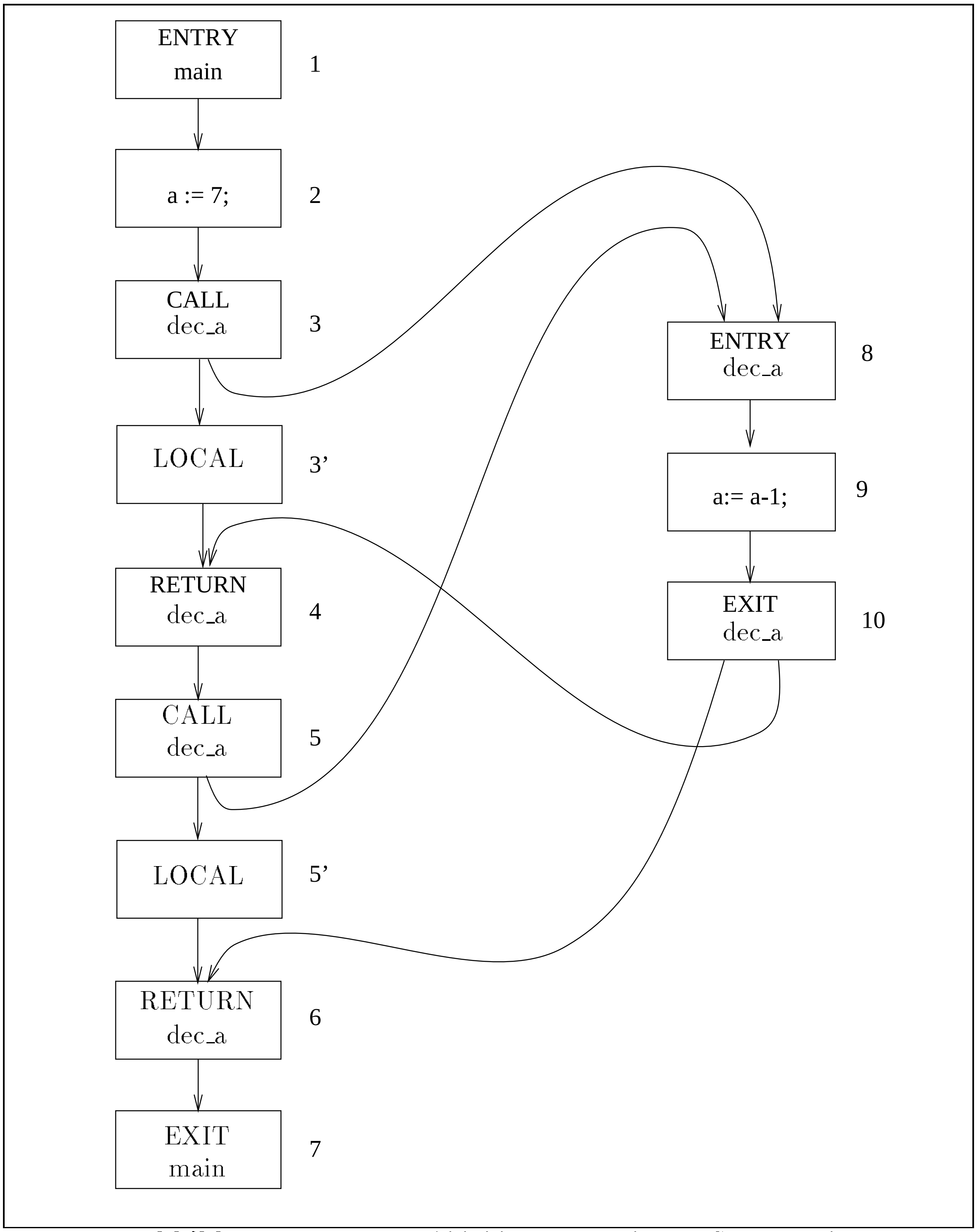

Abbildung 5.7: Der zu Abbildung 5.6 gehörige Supergraph

Allerdings ist durch eine einfache Änderung ein besseres Ergebnis zu erzielen. Man kann sehen, daß ein Supergraph, der mehr als eine Aufrufstelle für eine Prozedur P besitzt, einen Pfad von einer Aufrufstelle von P zu einem Return-Knoten zuläßt, der nicht zu diesem Aufrufknoten gehört. Der Pfad $\pi = 1, 2, 3, 8, 9, 10, 6, 7$ in Abbildung 5.7 ist z.B. kein gültiger Ausführungspfad. Um die Berechnung der Information entlang solcher Pfade zu verhindern, wird das Konzept der gültigen Pfade eingeführt. Ein Pfad heißt gültiger Pfad gleicher Ebene, wenn die Call– und Return-Knoten paarweise wie Klammern in einem geklammerten Ausdruck auftreten. Ein gültiger Pfad ist ein Teilstück eines gültigen Pfades gleicher Ebene. Eine solche Struktur läßt sich einfach durch eine kontextfreie Grammatik erzeugen.

Definition 5.2.2 (Gültiger Pfad)
Sei $G^* = (N^*, E^*, s^*, e^*)$ ein Supergraph, $\mathbf{P}[n_1, n_2]$ die Menge der Pfade von n_1 nach n_2 und P_W die durch die kontextfreie Grammatik $W = (\{A\}, N^*, \mathit{Prod}, A)$ definierte Menge, wobei *Prod* die folgenden Produktionen enthält:

A	$\rightarrow$	nA	für alle $n \in N^*$ ohne Call– und Return-Knoten
A	$\rightarrow$	AA	
A	$\rightarrow$	cAr	für zusammengehörige Call– und Return-Knoten c, r
A	$\rightarrow$	ϵ	wobei ϵ das leere Wort ist

Dann ist die Menge $P_S[n_1, n_2]$ der *gültigen Pfade gleicher Ebene* von n_1 nach n_2 definiert als

$$P_S[n_1, n_2] = P_W \cap P[n_1, n_2]$$

Ein *gültiger Pfad* von n_1 nach n_2 ist ein Teilstück eines gültigen Pfades gleicher Ebene von p nach q für beliebige Knoten p und q. Die Menge all dieser wird mit $P_G[n_1, n_2]$ bezeichnet oder auch nur mit $P[n_1, n_2]$. $\square$

Man kann also aus einem intraprozeduralen Datenflußproblem ein interprozedurales Datenflußproblem machen, indem man den nötigen Supergraphen berechnet und semantische Funktionen für die Call–, Local–, Return–, Entry–, und Exit-Knoten angibt, die die Effekte der Parameterübergabe und die Behandlung von lokalen Variablen festlegen.

Nun kann man interprozedurale Datenflußprobleme, ihre **MOP**– und **MFP**-Lösung definieren und wie im intraprozeduralen Fall die Koinzidenz und Korrektheit nachweisen. Dieser Beweis ist unter anderem in [Kno93] zu finden.

Ein weiteres Problem der globalen Analyse stellt das Modulkonzept dar. Die meisten größeren Programme referenzieren Prozeduren, deren Quelltext an einer anderen Stelle steht oder gar nicht bekannt ist, wie die Funktionen einer Bibliothek. Die globale Optimierung benötigt aber gerade eine globale Sicht aller Objekte und

Prozeduren, so daß der Programmierer immer versuchen sollte, den ganzen Quelltext für den Analysator zur Verfügung zu stellen.

Es gibt aber die Möglichkeit, eine Dummy-Prozedur einzuführen, die immer dann als aufgerufen angenommen wird, falls die tatsächlich referenzierte Funktion nicht verfügbar ist. Deren Prozedurrumpf kann dann eine Transferfunktion zugeordnet werden, die von Worst-Case Annahmen ausgeht, und so konservativ bleibt. Es ist aber auch möglich, andere Transferfunktionen zu benutzen, die das Verhalten der Prozedur exakter beschreiben, falls man dieses kennt. So könnte zu einer Bibliothek auch die Spezifikation der Transferfunktionen der einzelnen Funktionen geliefert werden, so daß die Bibliothek bei der globalen Optimierung kein Hindernis mehr darstellt.

Vorsicht ist geboten bei der getrennten Übersetzung einzelner Module, welche Funktionen enthalten, die von anderen Modulen aufgerufen werden können. Denn der Zweck der globalen Analyse besteht unter anderem darin, Informationen über alle Aufrufkontexte der Prozeduren aufzusammeln und mit diesem Wissen die Prozedur zu optimieren. Wenn eine so optimierte Prozedur aber nun aus einem Kontext heraus aufgerufen wird, der zur Analysezeit nicht bekannt war, so ist das Ergebnis nicht vorhersagbar. Falls man aber für alle Prozeduren aus einem Programm annimmt, daß sie von außerhalb her aufgerufen werden können, so verringern sich die möglichen Optimierungen erheblich.

Ein Kompromiß kann darin bestehen, die von außen direkt aufgerufenen Prozeduren speziell zu markieren[1] und dies bei der Optimierung entsprechend zu berücksichtigen.

Um das Verfahren der intraprozeduralen Analyse zu übertragen, ist es lediglich notwendig, den Supergraphen und die Call– und Return-Knoten zu kennen, um gültige Pfade konstruieren zu können.

5.2.3 Datenflußvektoren

Man benötigt für die Analyse einen Supergraphen und die Information, wie groß die Datenflußvektoren an jedem Knoten sein sollen. Dabei ist es nicht sinvoll, den Knoten einer Prozedur Datenflußvektoren verschiedener Längen zuzuordnen. Daher führt man eine Funktion **Arity** ein, die jeder Prozedur die Stelligkeit der Datenflußvektoren ihrer Knoten zuordnet. Man braucht außerdem an jedem Knoten, dessen Nachfolger eine andere Stelligkeit besitzt, eine Mapping-Relation, welche die verschiedenen Tupelpositionen aufeinander abbildet. Für zusammengehörige Call– und Return-Knoten müssen diese zueinander invers sein. Dies formalisiert die nachfolgende Definition:

[1] Z.B. in C können alle nicht aufgerufenen Funktionen als `static` markiert werden.

Definition 5.2.3 (Grapherweiterung)
Sei $G^* = (N^*, E^*, s^*, e^*)$ ein Supergraph für ein Programm P mit den Prozeduren $P_0, \cdots, P_k$ und $\mathbf{Arity} : \{P_i \mid 0 \leq i \leq k\} \rightarrow [1..]$ eine Funktion, die für jede Prozedur die Stelligkeit der zugehörigen Datenflußinformation festlegt. Dann heißt das Tripel $(G^*, \mathbf{Arity}, \mathbf{map}_c)$ *Grapherweiterung*, falls die $\mathbf{map}_c$ eine Familie von Funktionen sind, die für jede Aufrufstelle in P die Tupelpositionen der aufrufenden Funktion denen der aufgerufenen Funktion zuordnet, also für einen Aufruf $c = \text{Call } P_2$ in der Prozedur P_1 gilt: $\mathbf{map}_c : [1..\mathbf{Arity}(P_1)] \rightarrow [1..\mathbf{Arity}(P_2)]$. □

Wie bereits oben erwähnt, ist es sinnvoll zu fordern, daß die Stelligkeit für die Hauptprozedur immer eins ist. Daher wird im folgenden angenommen, daß

$$\mathbf{Arity}(P_0) = 1$$

gilt. Die Funktion **Arity** kann man konsistent auf Knoten aus N^* erweitern, so daß sie jedem Knoten ihre Stelligkeit zuordnet.

Statt nun den Supergraphen zusammen mit der **Arity** und dem $\mathbf{map}_c$ zur Iteration zu benutzen, definiert man einen erweiterten Supergraphen, in dem die Knoten Paare aus Knoten des Supergraphen und Tupelpositionen sind. Eine Kante von einem solchen Paar zu einem anderen existiert genau dann, wenn die Datenflußinformation des ersten Paares zu der des zweiten beiträgt. Dies führt zur folgenden Definition:

Definition 5.2.4 (Erweiterter Supergraph)
Sei $W = (G^*, \mathbf{Arity}, \mathbf{map}_c)$ mit $G^* = (N^*, E^*, s^*, e^*)$ eine Grapherweiterung. Dann ist $G^*_E = (N^*_E, E^*_E, s^*_E, e^*_E)$ der zu W gehörige *erweiterte Supergraph*, wenn gilt:

- $N^*_E = \{(n, i) \mid n \in N^* \text{ und } i \in [1..\mathbf{Arity}(n)]\}$,
- $s^*_E = (s^*, 1)$,
- $e^*_E = (e^*, 1)$ und
- $((n_1, i_1), (n_2, i_2)) \in E^*_E$ genau dann wenn $(n_1, n_2) \in E^*$ sowie eine der folgenden drei Bedingungen gilt:
    1. n_2 ist ein Entry-Knoten und $i_2 = \mathbf{map}_{n_1}(i_1)$
    2. n_1 ist ein Exit-Knoten und $i_1 = \mathbf{map}_c(i_2)$ wobei c der zu n_2 gehörige Call-Knoten ist
    3. $i_2 = i_1$ □

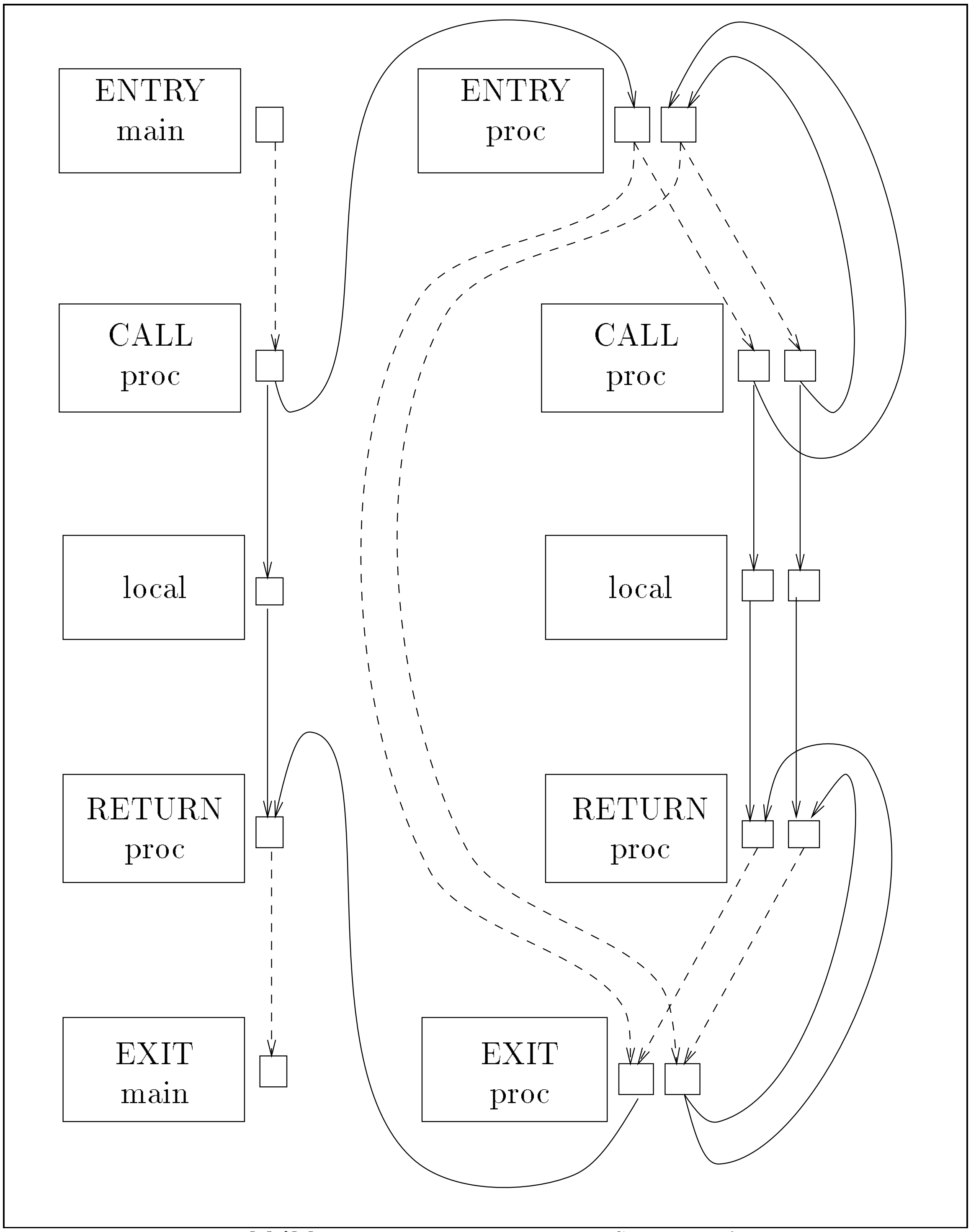

Abbildung 5.8: Ein erweiterter Supergraph

Ein Beispiel für einen Supergraphen ist in Abbildung 5.8 gezeigt.

Auf einen erweiterten Supergraphen können alle bekannten Fixpunktalgorithmen ohne irgend eine Veränderung angewandt werden. Die Vereinigung der Information für rekursive Prozeduren wird automatisch vorgenommen, wenn das mapping richtig gewählt wurde: Wenn eine Prozedur einen zweistelligen Datenflußvektor hat und sich selber aufruft, so wird man das mapping so wählen, daß beide Elemente auf ein Element des Vektors am Prozedurkopf abgebildet werden. Daher enthält der erweiterte Supergraph Kanten von den beiden Elementen an der Aufrufstelle zu einem Element am Kopf der Prozedur (vergleiche proc in Abbildung 5.8). Wenn die Information am Prozedureintritt neu bestimmt werden soll, wird der Iterationsalgorithmus zuerst die eingehende Information als die Vereinigung aller Vorgänger im Graph bestimmen (vergleiche Worklist Verfahren in Abbildung 4.2 auf Seite 34) und darauf die Transferfunktion des Prozedurkopfes anwenden. Die beiden Aufrufkontexte wurden also vereinigt.

Da die mapping Relationen an einem Prozedureintritt und –austritt invers zueinander sind, wird die Information des einen Datenflußelements am Prozedurende verdoppelt werden und zu dem rekusiven Aufruf propagiert.

5.2.4 Mappings

Die Aufgabe, ein geeignetes Paar von Mapping und Arity Funktionen auszuwählen, ist im wesentlichen gleichbedeutend mit der Aufgabe, einen Kompromiß zwischen Genauigkeit einerseits und Zeit- und Platzkomplexität der Analyse andererseits zu finden. Nun sollen einige Methoden, diese Funktionen zu berechnen, vorgestellt werden.

Einfachste Form

Die einfachste Art der interprozeduralen Analyse ist die, in der keine Datenflußvektoren sondern nur einstellige Informationen benutzt werden. Das kann simuliert werden, indem man wählt:

$$\forall n \in N^* : \mathbf{Arity}(n) = 1$$

$$\forall c : \mathbf{map}_c = id$$

So werden die Informationen aller Aufrufe einer Prozedur zusammengemischt, und eine Auftrennung findet nicht statt.

Universelle Methode

Eine andere Möglichkeit besteht darin, als Stelligkeit die Anzahl der Aufrufstellen einer Prozedur im Quelltext zu nehmen. Dies entspricht der Anzahl der eingehenden Kanten in den Entry-Knoten der Prozedur im Supergraphen. Allerdings muß die Stelligkeit der Hauptprozedur auf eine Zahl größer als null gesetzt werden, obwohl sie von keiner anderen Prozedur aufgerufen wird.

Für einen Aufruf $c =$ `Call p` in einer Prozedur q wählt man dann

$$\forall i \in [1..\mathbf{Arity}(q)] : \mathbf{map}_c(i) = x$$

für ein festes $x \in [1..\mathbf{Arity}(p)]$, das für jede Aufrufstelle verschieden gewählt wird.

Die Wirkung dieser $\mathbf{map}_c$ Funktionen ist, daß Datenflußinformationen jeder Aufrufstelle in der aufgerufenen Prozedur getrennt gehalten werden. Allerdings werden mehrere Datenflußelemente miteinander vermischt, falls es sich um einen mehrstufigen Aufruf handelt. Es werden also alle Aufrufketten der Länge eins getrennt behandelt.

Es hat sich bei praktischen Versuchen gezeigt, daß die mit diesem Verfahren berechnete Stelligkeit einiger Prozeduren in realen Programmen sehr groß (bis zu einigen hundert) werden kann. Das geschieht insbesondere dann, wenn sehr oft Prozedurvariablen benutzt werden. Es scheint also bei dem jetzigen Stand der Implementierung nicht sinnvoll, Methoden zu benutzen, die wesentlich längere Datenflußvektoren benutzen, da sonst die Dauer der Analysen und ihr Platzverbrauch zu groß werden.

Aufwendige Methode

Wird eine höhere Präzision als in der vorhergehenden Methode gewünscht, ist es notwendig, auch längere Aufrufketten getrennt zu analysieren. Gibt es z.B. zwei verschiedene Aufrufstellen c_1 und c_2 einer Prozedur p und p ruft q auf (siehe Beispiel 5.2.1 auf Seite 46), ist es nötig, mindestens so viele Felder im Datenflußvektor von q zu haben wie die Summe der Felder an c_1 und c_2, um jeweils beide Vektoren weiterhin getrennt behandeln zu können.

Die Stelligkeit der Datenflußelemente einer Prozedur sollte also gleich der Summe der Stelligkeiten aller Aufrufe sein. Als Basis für eine solche induktive Berechnung würde man die Stelligkeit von P_0 zu eins wählen. Dieses Verfahren läßt sich allerdings nur benutzen, wenn das vorliegende Programm nicht rekursiv, der Aufrufgraph also azyklisch ist.

Um dieses Problem zu umgehen, bestimmt man die starken Zusammenhangskomponenten[2] SZK's des Aufrufgraphen G_C und betrachtet diese als einzelne Knoten in einem Aufrufgraphen G'_C. Auf dem dadurch entstehenden azyklischen Graphen G'_C kann man nun die oben beschriebene induktive Berechnung durchführen. Allerdings multipliziert man die so berechnete Stelligkeit eines Verbundknoten noch mit einer Konstanten K, um die potentiell unendliche Anzahl an Aufrufen innerhalb einer solchen Verbundkomponente zu respektieren. Anschließend werden die verschmolzenen Knoten wieder expandiert, und jeder enthaltenen Prozedur wird die für den Verbundknoten berechnete Stelligkeit zugewiesen.

Das resultierende Mapping für Aufrufe, die von einer SZK in einen einfachen Knoten führen oder umgekehrt, ist einfach, da es für jedes Element im Vektor der Aufrufstelle einen Platz im Vektor der aufgerufenen Prozedur gibt. Für die übrigen Mappings gibt es verschiedene Wahlmöglichkeiten.

Die Berechnung der Stelligkeit nach dieser Methode ist in Abbildung 5.9 dargestellt.

5.2.5 Zusammenfassung

Nachfolgend sollen die wichtigsten Vor– und Nachteile der verwendeten Technik zur interprozeduralen Analyse noch einmal zusammengefaßt werden:

— Die berechnete Lösung ist nicht präzise, auch wenn der zugrunde liegende Verband endlich ist. Dies kann allerdings durch eine andere Implementierung vermieden werden. Die in Kapitel 5.1.3 vorgestellte Technik ermöglicht dies.

\+ Jedes intraprozedurale Datenflußproblem kann mit der vorliegenden Methode interprozedural erweitert werden, ohne daß zusätzliche Beschränkungen des Verbandes nötig wären, um die Terminierung zu garantieren.

\+ Um eine solche Erweiterung vorzunehmen, ist es nur nötig, zusätzlich Mechanismen für die Parameterübergabe, Rückgabewerte und lokale Variablen zu spezifizieren.

\+ Eine Abstimmung zwischen Genauigkeit und Platz- und Zeitverbrauch der Analyse kann einfach mittels der **Arity** und $\mathbf{map}_c$ Funktionen vorgenommen werden.

\+ Man erhält genaue Ergebnisse über verschiedene Aufrufe einer Prozedur.

[2] Zur genauen Definition und Algorithmen zur Berechnung siehe [Meh84].

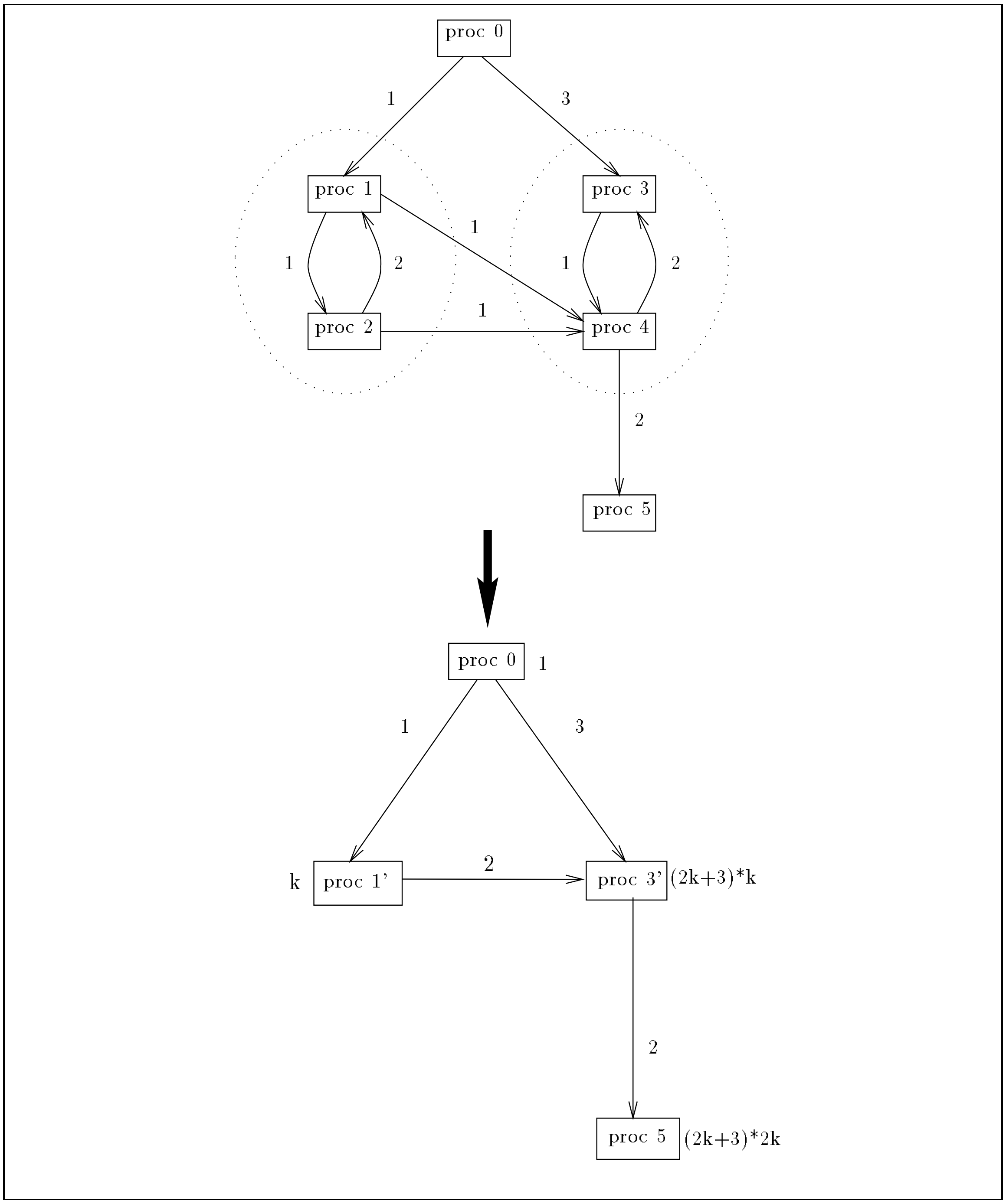

Abbildung 5.9: Die Berechnung der Stelligkeit nach der aufwendigen Methode in 5.2.4. Die Zahlen an den Kanten sind die Anzahl der Aufrufstellen in der jeweiligen Prozedur, und die Zahlen an den Knoten sind die berechneten Stelligkeiten, d.h. die Kante von `proc 2` zu `proc 1` bedeutet, daß es zwei Aufrufstellen von `proc 2` in `proc 1` gibt.

Kapitel 6

Generierung von Analysatoren

Werden die in Kapitel 4 und 5 vorgestellten Lösungsalgorithmen für mehrere verschiedene Datenflußprobleme implementiert, so fällt auf, daß sich die Programme in großen Teilen sehr ähnlich sind. Z.B. ist die Lösungsprozedur mittels einer Workinglist bis auf einige Funktionsaufrufe und Namen von Datentypen immer identisch und sieht so wie der in Abbildung 4.2 abgedruckte Pseudocode aus.
Dies legt nahe, die immer gleichen Teile eines Analysators einmal zu programmieren und zu parametrisieren. Aber auch andere Teile des Analysators können systematisch von einem Generator aus einer Spezifikation erzeugt werden.

Um eine (interprozedurale) Datenflußanalyse festzulegen, sind folgende Komponenten nötig:

1. Ein vollständiger Verband V
2. Eine meta Transferfunktion, die angewandt auf einen Knoten eines Kontrollflußgraphen eine (monotone) Transferfunktion von V nach V ergibt
3. Ein Fixpunktalgorithmus mit den zugehörigen Datenstrukturen, wie z.B. ein iterativer Algorithmus mit einer Workinglist
4. Ein Kontrollflußgraph

Bei diesen Punkten sind 1. bis 3. zur Generierungszeit eines Analysators bekannt, während der Kontrollflußgraph erst zur Laufzeit des Analysators bekannt ist.

Für PAG sind spezielle Sprachen entwickelt worden, die der Spezifikation der obigen Punkte dienen. Eine Spezifikation besteht aus folgenden Teilen:

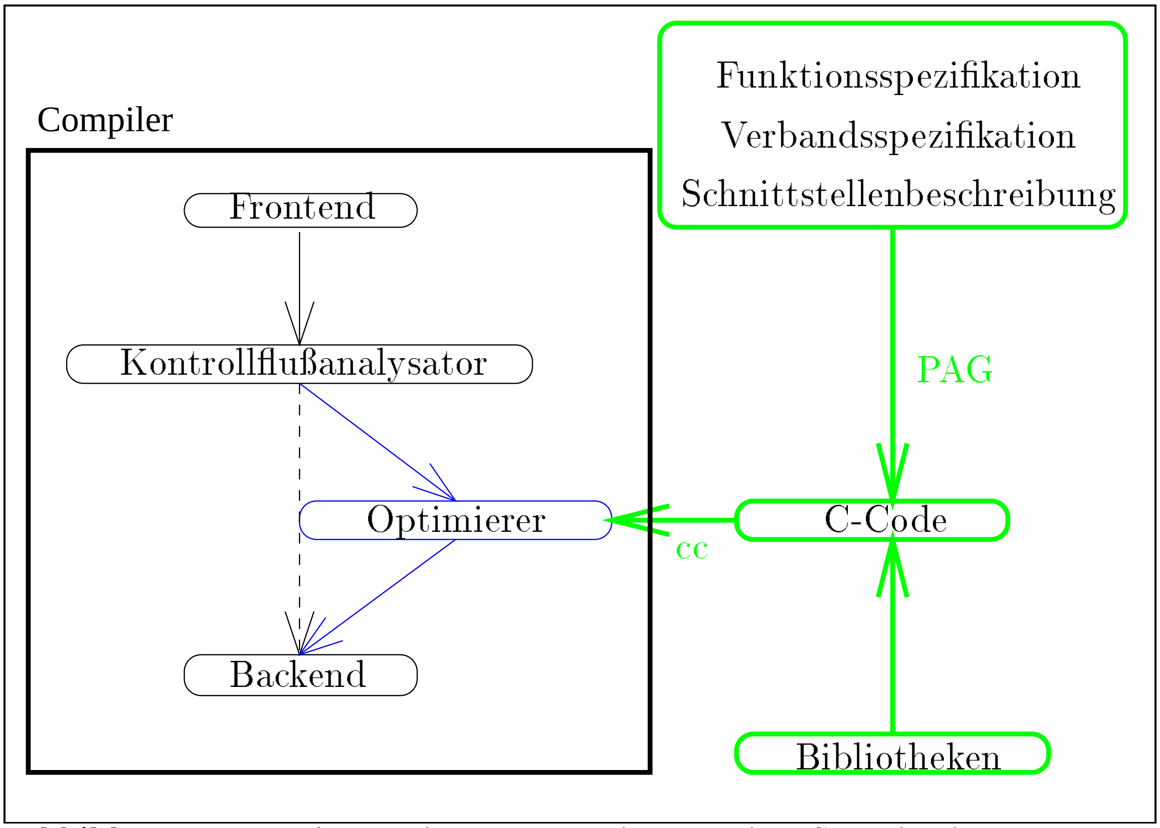

Abbildung 6.1: Laufzeit und Zusammenarbeit mit dem Compiler der generierten Analysatoren

1. Der Spezifikation des zugrunde liegenden Verbandes in DATLA,
2. der Beschreibung der Transferfunktionen in FULA und
3. der Nennung der Lösungsmethode.

Die einzelnen Sprachen werden in Kapitel 7 beschrieben.

Die Idee, Datenflußanalysatoren zu generieren anstatt sie jedesmal aufs neue von Hand zu implementieren, ist nicht neu. Es hat eine Reihe von Experimenten mit generierenden Systemen gegeben, aber nur wenige waren erfolgreich, da die praktischen und theoretischen Probleme trotz der scheinbaren Klarheit zahlreich sind. So ist es von entscheidender Bedeutung, daß einerseits die Spezifikationssprachen mächtig genug sind, um eine große Klasse von Problemen abdecken zu können und daß andererseits der generierte Code effizient ist. Das ist besonders wichtig, wenn Analysatoren für den Einsatz in Compilern und nicht nur zum Testen der Analyse generiert werden sollen.

6.1 Die Struktur von PAG

PAG ist entworfen worden, um Analysatoren für Übersetzungssysteme zu generieren. Die Hauptannahme dabei ist, daß solch ein Übersetzungssystem ein Interface zu dem Syntaxbaum und dem Kontrollflußgraphen des zu analysierenden Programms anbietet. Solch eine Repräsentierung kann in fast jedem Compiler gefunden werden.

Das Interface zum Kontrollflußgraphen muß Funktionen anbieten, die es ermöglichen, den Graphen zu traversieren, auf den Identifier eines Knoten zuzugreifen und den Syntaxbaum zu erreichen, mit dem ein Knoten beschriftet ist. Um die Implementierung der Datenstrukturen des Graphen von ihrer logischen Funktionalität zu trennen, muß der Designer einer Analyse ein Interface schreiben, das von allen generierten Analysatoren für dieses generierende System benutzt werden kann.

Es ist notwendig für PAG, die Struktur des abstrakten Syntaxbaumes zu kennen, um die notwendigen Zugriffsfunktionen generieren zu können. Diese kann durch eine erweiterte Baumgrammatik definiert werden. Daher benötigt der Generator ein Eingabefile, welches die Struktur des Baumes beschreibt.

Weiterhin gibt es Eingabefiles, um den unterliegenden Verband, die Transferfunktionen und globale Parameter des Analysators zu beschreiben.

Abbildung 8.1 auf Seite 78 zeigt einen Überblick über PAG während in Abbildung 6.1 das Zusammenspiel der generierten Analysatoren mit dem Compiler zu sehen ist.

Kapitel 7

Die Spezifikationssprachen

Die Spezifikation eines Datenflußproblems gliedert sich in vier verschiedene Teile. Diese dienen zur Spezifikation

- der allgemeinen Problemcharakteristik
- des Aufbaus des Syntaxbaumes
- des zugrunde liegenden Verbandes u
- der Transferfunktionen.

In der allgemeinen Problemcharakteristik werden verschiedene Parameter des zu generierenden Analysators angegeben. Dies sind z.B. die Richtung des Problems (vorwärts oder rückwärts) und der zu verwendende Lösungsalgorithmus.

Der Aufbau des Syntaxbaumes muß, wie bereits erwähnt, bekannt sein, damit die Transferfunktionen in Abhängigkeit des Teilbaumes spezifiziert werden können, mit denen der betreffende Knoten des Kontrollflußgraphen beschriftet ist. Es handelt sich hierbei um die einfache Aufzählung der Signatur der vom Frontend erzeugten Bäume.

Der zugrunde liegende Verband wird induktiv aufgebaut aus Basiselementen (wie z.B. der vordefinierten Menge der ganzen Zahlen) mit verschiedenen Operatoren (wie z.B. der Produkt- oder Funktionsraumbildung).
Die Transferfunktionen werden schließlich in einer funktionalen Sprache beschrieben, die es ermöglicht, die Funktionen durch Patternmatching über die Struktur des Syntaxbaumes zu beschreiben. Auf alle vier Teile dieser Spezifikation soll nun näher eingegangen werden. Ein Beispiel einer vollständigen Spezifikation eines Analysators ist in Anhang A zu finden. Es handelt sich um einen Konstantenpropagator für

die Zwischensprache CCMIR (siehe [COM93a]), mit der unter anderem ANSI C analysiert werden kann.

7.1 Notationen

Im folgenden werden reservierte Worte und Symbole **fett** geschrieben, Nichtterminale in spitzen Klammern $\langle\rangle$ und Eingaben in `Schreibmaschinenschrift`. Bei den Nichtterminalen wird unterschieden zwischen Typen im allgemeinen $\langle type\rangle$, vordefinierten Typen $\langle predef\_type\rangle$, syntaktischen Typen $\langle synt_type\rangle$und benutzerdefinierten Typen $\langle user\_type\rangle$. Außerdem gibt es Pattern $\langle pat\rangle$, Ausdrücke $\langle exp\rangle$, Ausdrücke mit einem boolschen Wert $\langle bool\_exp\rangle$, Variablen $\langle var\rangle$ und Konstanten $\langle const\rangle$.

Zur Gruppierung von Symbolen werden geschweifte Klammern $\{\}$ verwendet, Alternativen werden mit $|$ gekennzeichnet und Wiederholungen durch hochgestellten * oder $^+$.

7.2 Syntaxbaum

Der Aufbau des abstrakten Syntaxbaums wird mittels einer erweiterten Baumgrammatik angegeben. Da Listen von syntaktischen Kategorien wesentliche Elemente von Programmiersprachen darstellen, wie z.B. eine Liste von Anweisungen, sind diese zur Vereinfachung vordefiniert. Außerdem können in der Definition Annotierungen vorgenommen werden, um Konstrukte zu deklarieren, die bei der Datenflußanalyse gesondert behandelt werden sollen, wie die Deklarationen von Prozeduren, oder Anweisungen, die Ausnahmen auslösen können, oder die einen Call–, Local–, Return–, Entry– oder Exit-Knoten darstellen sollen.

Eine Sorte[1] s wird definiert durch die Aufzählung der Operationen, deren Ergebnistyp s ist. Bei den Operatoren[2] wird der Typ ihrer Argumente mitangegeben. Außerdem erhält jedes Argument einen Namen, der den Namen der Zugriffsfunktionen mitbestimmt. Der Name der Funktion ergibt sich als

$$\langle Sortenname\rangle\_\langle Operatorname\rangle\_\mathrm{get}\_\langle Feldname\rangle.$$

Diese Funktionen muß das Interface zur Verfügung stellen. Alle Sorten und Listen von Sorten, die hier definiert werden, sind Typen, die in FULA benutzt werden können.

[1] Eine Sorte wird im folgenden auch syntaktischer Typ genannt

[2] bzw. Konstruktoren

Beispiel 7.2.1
Hier wird eine Sorte `Statement` mit den Operatoren `If` und `Sequenz` definiert. Der Typ von `If` ist `Expression, Statement, Statement` → `Statement`. Der Stern `*` ist der oben erwähnte Listenoperator.

```
Statement : If(Cond:Expression,Then:Statement,Else:Statement)
          | Sequenz(List:Statement*);
```

Um auf den Then-Teil eines Statements zuzugreifen, benutzt PAG die Funktion

`Statement_If_get_Then()`.

□

Einzelne Operatoren können mit Annotierungen versehen werden. Dies sind im einzelnen:

1. **CALL**, **LOCAL** und **RETURN** um einen Call–, Local– bzw. Return– Knoten zu identifizieren
2. **BEGIN**, **END** um die Anfangs– und End-Knoten von Prozeduren zu markieren
3. **EXCEPTION** um anzuzeigen, daß von dieser Anweisung Ausnahmen ausgelöst werden können. Diese Möglichkeit wird aber bislang nicht unterstützt.

Schließlich ist es ebenfalls notwendig anzugeben, daß gewisse Typen des Syntaxbaumes in Standardtypen umgewandelt werden können. Damit kann gewährleistet werden, daß z.B. eine Sorte `Integer` auch in den vordefinierten Typ der ganzen Zahlen konvertiert und weiter verarbeitet werden kann. Die vordefinierten Typen werden bei der Verbandsspezifikation näher erläutert. Eine solche Typäquivalenz wird durch Regeln der Form

$$\langle \textit{synt\_type} \rangle \texttt{ == } \langle \textit{predef\_type} \rangle$$

angegeben. PAG generiert für jede solche Regel cast Funktionen für FULA, um den syntaktischen Typ in den vordefinierten Typ umzuwandeln.

7.3 Problemspezifikation

In diesem Teil der Spezifikation werden Angaben über die allgemeine Natur des Problems gemacht und eventuell Lösungsstrategien vorgegeben.

Dazu gehört die Angabe, ob es sich um ein Vorwärts– oder ein Rückwärtsproblem handelt. Außerdem ist hier der Name des Trägerverbandes anzugeben, der in DATLA definiert wurde.
Damit festgelegt werden kann, ob es sich um ein Vereinigungs– oder ein Schnittproblem handelt, kann hier auch angegeben werden, mit welchem Element des Verbandes die Knoten zu Beginn zu initialisieren sind und wie die Informationen aus mehreren eingehenden Kanten miteinander kombiniert werden sollen. Die letzten beiden Angaben werden als Ausdrücke der Sprache FULA angegeben, die auch der Spezifikation der Transferfunktionen dient.
Schließlich kann, ebenfalls in FULA, ein spezieller Initialisierungswert für den Startknoten angegeben werden. Damit hat der Benutzer weitreichende Möglichkeiten, die Natur des zu lösenden Problems anzugeben.

An dieser Stelle kann optional auch eine der im vorangegangenen Kapitel ausführlich erläuterten Lösungsstrategien, wie Worklistiteration, Nielson Iteration oder Widening and Narrowing angegeben werden.

7.4 Mengen- und Verbandsspezifikation (DATLA)

Hier wird der dem Datenflußproblem zugrunde liegenden Verband beschrieben. Da es sich hierbei um die Beschreibung von Datentypen handelt, heißt die verwendete Sprache DATLA, eine Abkürzung für Datatype Definition Language. Die Spezifikation ist in zwei Teile geteilt: Einen zur Definition von Mengen und einen zur Definition von Verbänden. Die Typen werden bottom up aufgebaut, d.h. aus kleineren Elementen, wie den vordefinierten Typen, werden sukzessive größere und kompliziertere Typen aufgebaut.

Im Gegensatz zu vielen anderen Generatoren ist es in DATLA möglich, Mengen und Verbände simultan rekursiv zu definieren. Dies gibt dem Benutzer eine große Anzahl von Möglichkeiten, die benötigten Typen zu definieren. Aber diese und andere Möglichkeiten von DATLA führen dazu, daß Mengen und Domains mit unendlich vielen Elementen und sogar mit unendlich langen Ketten definiert werden können. Daher ist es die Aufgabe des Benutzers, dafür zu sorgen, daß nur ein endlicher Teil des Verbandes benutzt wird oder die Terminierung auf eine andere Art sicherzustellen.

Umgebungen sind ein Beispiel, in dem nur ein endlicher Teil eines unendlichen

Verbandes benutzt wird. In einer typischen Programmanalyse wird an jedem Programmpunkt für jede Variable, die im Quelltext vorkommt, ein Wert gespeichert. Dies wird üblicherweise durch eine Funktion von Variablen in die gewünschte Information modelliert. Da die Menge aller Variablen unendlich ist, gibt es in diesem Verband unendlich lange aufsteigende Ketten. Aber weil in jedem Programm nur eine endliche Teilmenge aller Variablen benutzt wird, ist die Terminierung garantiert.

7.4.1 Mengenspezifikation

Der Teil zur Definition von Mengen wird mit dem Schlüsselwort **SET** eingeleitet. Um einfach auf gewisse Grundtypen zugreifen zu können, sind folgende Mengen bereits vordefiniert:

1. **snum**: Die Menge der ganzen Zahlen
2. **unum**: Die Menge der positiven ganzen Zahlen (vorzeichenlos)
3. **real**: Alle Gleitkommazahlen
4. **bool**: Die Werte **true** und **false**
5. **chr**: Alle Ascii-Zeichen
6. **string**: Zeichenketten aus Ascii-Zeichen

Außer diesen Mengen können durch Aufzählung beliebige andere (endliche) Mengen definiert werden.

Die Liste der vordefinierten Mengen kann bei Bedarf leicht durch zusätzliche Elemente erweitert werden. Dazu muß für diese Typen eine festgelegte Anzahl von Funktionen definiert werden.

Schließlich kann eine neue Menge definiert werden, indem ein Mengenoperator auf eine andere Menge angewandt wird. Dazu stehen folgende Mengenoperatoren zur Verfügung:

1. disjunkte Summe endlich vieler Mengen: Dies ist die Vereinigung dieser Mengen, bei der gleiche Elemente verschiedener Mengen unterschieden werden. Ein Element, das in zwei der Mengen vorkommt, wird also durch ein *Tag* unterschieden und so zweimal zur resultierenden Menge hinzugefügt.
2. endliche Tupelbildung

3. Potenzmengenbildung: Es wird die Menge aller Teilmengen von Elementen der originalen Menge gebildet.

4. Listen von Elementen einer Menge

5. Bildung des Funktionsraumes von einer Menge M_1 in eine Menge M_2

Wie leicht zu sehen ist, sind die durch diese Sprache definierbaren Mengen nicht endlich und unterliegen auch sonst kaum einer Beschränkung.

7.4.2 Verbandsspezifikation

Der Teil zur Definition von Verbänden beginnt mit dem Wort **DOMAIN**. Hier werden Verbände konstruiert, zum Teil unter Zuhilfenahme der im ersten Teil definierten Mengen.

Es gibt folgende Möglichkeiten, einen Verband anzugeben:

1. Die flache Anordnung einer Menge oder eines Verbandes, wie sie in Kapitel 3 beschrieben und in Abbildung 3.4 auf Seite 13 dargestellt ist.

2. Die Bildung des Potenzverbandes aus einer Menge oder eines Verbandes, wie in Satz 3.1.7 (Seite 15)

3. Das Liften eines Verbandes (vgl. Satz 3.1.8, Seite 16)

4. Die Tupelbildung endlich vieler anderer Verbände (Satz 3.1.5, Seite 15)

5. Die Bildung des Funktionsraumes einer Menge oder eines Verbandes in einen Verband aus Satz 3.1.9 (Seite 16) und

6. durch Aufzählung der Elemente mit expliziter Angabe einer vollständigen Ordnung. Das System überprüft dabei, ob es sich tatsächlich um einen vollständigen Verband handelt.

7.4.3 Anwendungen

Um die Mächtigkeit der Sprache DATLA zu demonstrieren, seien hier einige kurze Beispiele gegeben.

Zuerst soll gezeigt werden, wie man eine Liste von ganzen Zahlen definieren kann. Dabei soll der vordefinierte Operator **list** nicht verwendet werden, um die Möglichkeiten des Programmierers zu verdeutlichen.

Beispiel 7.4.1

```
SET
E    = { empty }
T    = snum * LIST
LIST = E ++ T
```

In der ersten Zeile wird `E` als eine Menge definiert, die nur ein Element, nämlich `empty` besitzt. Anschließend wird ein Tupel aus der vordefinierten Menge `snum` und einer Liste mit Namen `T` definiert.
Schließlich ist eine Liste eine disjunkte Summe aus der leeren Liste und einem Paar aus einer Zahl und einer Restliste.

Die definierte Menge ist also

$$\texttt{LIST} = \{\texttt{empty}, (z, \texttt{empty}), (z, (z, \texttt{empty})), (z, (z, (z, \texttt{empty}))), \cdots\}$$

wobei z jeweils eine beliebige Zahl aus `snum` ist. □

Ebenso können binäre Bäume mit Zahlen als Knotenbeschriftung durch folgenden Code kreiert werden.

Beispiel 7.4.2

```
SET
NODE  = LEAF ++ INNER
LEAF  = snum
INNER = snum * NODE * NODE
```

□

Auch ganze Graphen können konstruiert werden. Diese bilden, im Unterschied zu den vorherigen Beispielen, nicht nur eine Menge sondern einen Verband.

Beispiel 7.4.3

```
SET
NODE  = snum          // Knoten sind mit Zahlen beschriftet
EDGE  = NODE * NODE   // Kanten sind Paare aus Knoten

DOMAIN
PEDGE = power(EDGE)   // Die Menge aller Kanten
PNODE = power(NODE)   // Die Menge der Knoten
GRAPH = PNODE * PEDGE // Graphen sind Paare aus Knoten und Kanten
```

Da die implizit definierte Ordnung nicht struktureller Natur ist (zwei Knoten sind nur gleich, wenn sie mit der gleichen Zahl beschriftet sind), ist bei der Benutzung darauf zu achten, daß Knoten gleicher Bedeutung mit derselben Zahl beschriftet werden. □

7.5 Spezifikation des Datenflußproblems (FULA)

Dieser Teil besteht wiederum aus zwei Teilen, die jedoch auf der gleichen Spezifikationssprache beruhen. Im ersten Teil wird die eigentliche Transferfunktion beschrieben, während im zweiten Teil Hilfsfunktionen definiert werden können.
Bei der verwendeten Sprache handelt es sich um eine eigens für diesen Zweck entworfene funktionale Sprache erster Ordnung. Die Sprache wurde so entworfen, daß sich die Sprachkonstrukte effizient in die imperative medium-level Sprache C übersetzen lassen. Zu diesem Zweck wurden einige Einschränkungen gegenüber bekannten funktionalen Sprachen wie HOPE hingenommen, um den als groß erachteten Nutzen der Effizienz und Portabilität von C nutzen zu können.
FULA läßt sich durch folgende Schlagworte gut beschreiben:

- funktionale Sprache 1. Ordnung
- eager evaluation
- streng getypt
- monomorph
- statische Sichtbarkeitsregeln
- Typinferenz
- benutzterdefinierte Typen
- kein expliziter Fixpunktoperator

7.5.1 Allgemeine Struktur

Die Grundelemente von FULA sind Ausdrücke. Sie werden eager ausgewertet, d.h. bevor ein Ausdruck ausgewertet wird, werden alle seine Teilausdrücke ausgewertet. Sie haben einen eindeutigen Typ, der statisch bestimmt werden kann. Das wird durch den Typinferenzalgorithmus vorgenommen. Falls dieser allerdings nicht in der Lage ist, einen vollständig spezialisierten Typ für einen Ausdruck herzuleiten, kann der Benutzer den Typ eines Ausdrucks durch

$$\boxed{\langle type\rangle \ \$ \ \langle exp\rangle}$$

angeben. Dieser Ausdruck ist nicht mit einem Typecast zu verwechseln, da hier der Typ eines Ausdrucks nicht verändert werden kann, sondern er nur eine Hilfe zur Bestimmung des Typs ist.

Es werden keine impliziten Typumwandlungen vorgenommen. Falls eine Typumwandlung benötigt wird, stehen explizite Operatoren zur (sinnvollen) Typumwandlung zur Verfügung.

Variablen gehorchen in FULA statischen Sichtbarkeitsregeln. Die zugehörige Definition für ein angewandtes Vorkommen einer Variablen v ist das lexikalisch innerste bindende Konstrukt, in dem die Variable definiert wird. Bindende Konstrukte sind der **let**-Ausdruck, die Funktionsdefinition und der **case**-Ausdruck.

7.5.2 Transferfunktionen

Der FULA Code spaltet sich in zwei Abschnitte auf: Einen, um Hilfsfunktionen zu definieren und einen für die Transferfunktionen selbst. Diese Transferfunktionen werden in einer speziellen Notation geschrieben: Sie haben keinen Namen und werden mittels Patternmatching über die Beschriftung der Kontrollflußknoten definiert. Außerdem haben sie einen impliziten Parameter namens `@`, der den eingehenden Datenflußwert enthält. Der Rückgabewert muß der ausgehende Datenfluß sein.

7.5.3 Funktionen

In FULA werden zwei verschiedene Arten von Funktionen unterschieden. Die einen werden extensional beschrieben und entsprechen einem Datentyp. Ihr Typ muß im entsprechenden Teil der Spezifikation beschrieben werden. Die andere Art sind die der intensional in FULA definierten Funktionen, die einem Stück Programmcode entsprechen. Die erste Art von Funktionen kann als Argument der zweiten Art auftauchen, und da es sich um eine Sprache erster Ordnung handelt, können die als Programmcode beschriebenen Funktionen nicht als Argumente benutzt werden.

Hier soll nun auf die FULA-Funktionen (der zweiten Art) eingegangen werden. Eine Funktion kann nur mit einem Namen versehen definiert werden, d.h. es existiert keine Lambda Abstraktion. Eine solche Definition einer Funktion f kann fallweise und beliebig verteilt über den Quelltext erfolgen, indem Pattern für die Argumente benutzt werden. Ein einzelner Fall hat folgende Syntax:

$$\boxed{f(\langle pat\rangle^*) = \langle exp\rangle;}$$

Für jede Funktion kann optional eine Typdeklaration angegeben werden:

$$\boxed{f :: \langle type\rangle^* -> \langle type\rangle;}$$

Geschachtelte Funktionsdefinitionen sind nicht zulässig.

7.5.4 Pattern

Pattern werden für die fallweise Definition von Funktionen benutzt. Sie können aber auch in **case**-Ausdrücken verwendet werden. Sie können verschachtelt auftreten und binden Variablen. Folgende Elemente sind zulässig, wobei allerdings semantische Nebenbedingungen zu erfüllen sind, da nicht alle Patternarten für alle Typen zulässig sind.

- $\boxed{\imath}$ Elemente einer aufgezählten Menge bzw. eines Domains
- $\boxed{\langle const\rangle}$ Konstanten der eingebauten Typen wie **true**, `1.2` und '`c`'
- $\boxed{\top}$ oder $\boxed{\bot}$ Top– und Bottomelemente von Domaintypen
- $\boxed{\{\ \}}$ Leere Menge
- $\boxed{\langle var\rangle}$ Variablen. Diese dürfen allerdings pro Pattern nur einmal verwendet werden, d.h. es sind nur lineare Pattern zulässig
- $\boxed{\_}$ Namenlose Variablen, die keinerlei Bindung erzeugen
- $\boxed{[\,]}$ oder $\boxed{\langle pat\rangle :: \langle pat\rangle}$ Conspattern für Listen, die in DATLA deklariert wurden
- $\boxed{[\,!\,]}$ oder $\boxed{\langle pat\rangle :: \langle pat\rangle}$ Conspattern für Listen von syntaktischen Typen, die im Syntax-Teil der Spezifikation durch einen Stern deklariert wurden
- $\boxed{\langle constructor\rangle\ (\ \langle pat\rangle^*)}$ syntaktische Konstruktoren, die im Syntax-Teil definiert wurden
- $\boxed{(\langle pat\rangle, \langle pat\rangle, \cdots)}$ Tupelpattern

Zusätzlich kann zu einem Pattern optional, wie auch bei Ausdrücken, der Typ mit angegeben werden:

$$\boxed{\langle type\rangle\ \$\ \langle pat\rangle}$$

7.5.5 Ausdrücke

Folgende Ausdrücke sind in FULA definiert:

1. Konstanten (eingebaute sowie vordefinierte).

2. $\boxed{\textbf{if}\ \langle \textit{bool\_exp}\rangle\ \textbf{then}\ \langle \textit{exp}_1\rangle\ \textbf{else}\ \langle \textit{exp}_2\rangle\ \textbf{endif}}$ Konditionale, um den Wert eines Ausdrucks in Abhängigkeit eines anderen zu definieren. Dies kann als das einzige Konstrukt der Sprache angesehen werden, dessen Auswertung nicht eager ist, da zuerst $\langle \textit{bool\_exp}\rangle$ ausgewertet wird und anschließend entweder $\langle \textit{exp}_1\rangle$ oder $\langle \textit{exp}_2\rangle$, aber niemals beide.

3. $\boxed{\textbf{let}\ \{\langle \textit{var}_i\rangle\ \{ = \mid <= \}\ \langle \textit{exp}_i\rangle\ ;\ \}^+\ \textbf{in}\ \langle \textit{exp}\rangle}$ um eine Anzahl von Variablen, nämlich die $\langle \textit{var}_i\rangle$ lokal zum Ausdruck $\langle \textit{exp}\rangle$ einzuführen. Diese werden an die Werte von den $\langle \textit{exp}_i\rangle$ gebunden und sind nur in $\langle \textit{exp}\rangle$ sichtbar. Für Variablen, die mit einem `<=` zugewiesen wurden, wird nach der Auswertung des zugehörigen $\langle \textit{exp}_i\rangle$ geprüft, ob der Wert $\top$ oder $\bot$ ist, und dies dann als Wert des gesamten let Ausdrucks verwendet. Diese Notation dient nur der Übersichtlichkeit. Bei der Auswertung wird zuerst jedes $\langle \textit{exp}_i\rangle$ ausgewertet, bevor der Wert von $\langle \textit{exp}\rangle$ bestimmt wird.

4. $\boxed{\textbf{case} \left\{\langle \textit{var}_j\rangle\right\}^+ \textbf{of} \left\{\langle \textit{pat}\rangle^+ => \langle \textit{exp}_i\rangle\ \right\}^+ \textbf{endcase}}$ gibt die Möglichkeit, die Struktur eines oder mehrerer Ausdrücke mittels Patternmatching zu untersuchen, die an die $\langle \textit{var}_j\rangle$ gebunden sind. Das Ergebnis dieses Ausdrucks ist der Wert des ersten $\langle \textit{exp}_i\rangle$ für das alle $\langle \textit{pat}_j\rangle$'s mit den Werten der $\langle \textit{var}_j\rangle$'s in Übereinstimmung gebracht werden können.

5. Funktionsanwendungen: Hierbei sind beide Arten von Funktionen gemeint, sowohl DATLA als auch FULA Funktionen. Die Anzahl der Argumente muß der Stelligkeit der Funktion entsprechen. Andernfalls wäre das Ergebnis eines solchen Ausdrucks funktional, was nicht zulässig ist.

6. $\boxed{\textbf{print}(\langle \textit{exp}_1\rangle) ++ \langle \textit{exp}\rangle}$ ist äquivalent zu dem Ausdruck $\langle \textit{exp}\rangle$, mit dem Seiteneffekt, daß $\langle \textit{exp}_1\rangle$ ausgegeben wird.

7. Anwendungen von eingebauten Funktionen in Prefix oder Infix Notation. Viele dieser Funktionen werden für bestimmte Datentypen generiert, z.B. Projektionsfunktionen für Tupeltypen. Folgende Funktionen werden generiert:

   - für jeden Typ: gleich `=` und ungleich `!=`
   - für Domaintypen: kleinste obere Schranke **lub**, größte untere Schranke **glb**, größer $>$, größer gleich $>=$, kleiner $<$ und kleiner gleich $<=$

- für **unum**, **snum**, **real**:
  (a) mathematische Operationen: Addition `+`, Subtraktion `-`, Multiplikation `*`, ganzzahlige Division `/`, Restbildung `%` und Betrag $|.|$
  (b) bitweise Operationen: Und `&`, Oder `|` und exklusives Oder `^`
  (c) Vergleichsfunktionen: größer $>$, größer gleich $>=$, kleiner $<$ und kleiner gleich $<=$
  (d) Cast Funktionen: **snum**, **unum**, **chr**, **real**
- für **bool**: Und `&&`, Oder `||` und Nicht `!`
- **chr**: cast Funktionen **snum**, **unum**, **real**
- **string**: Konkatenation `.`, Auswahl eines Zeichens $[.]$
- flache oder geliftete Domains: **lift**, **drop**
- Listen: cons ‚:'
- Summentypen: **is**-$\langle$*type_name*$\rangle$, **down**-$\langle$*type_name*$\rangle$, **up**-$\langle$*type_name*$\rangle$, um ein Element einer Summenkomponente in die Summe zu heben und umgekehrt
- Tupel: Projektion `!`, Tupel Konstruktion $(\langle exp_1\rangle, \cdots, \langle exp_n\rangle)$
- Power Mengen/Domains: Zufügen eines Elementes `~`, Entfernen eines Elementes `#`, ist-Element-von Test `??`
- Funktionstypen: Anwendung $.\{.\}$, Wert Änderung `.[.\.]`, Erzeugung mit einem Default Element $[->\ .]$

Kapitel 8

Implementierung

Die grundlegenden Techniken und Algorithmen, die in PAG verwendet werden, wurden in den Kapiteln 4, 5 und 6 ausführlich erläutert. Nun sollen einige Teile genauer beleuchtet werden. Ein Überblick über das Gesamtsystem ist in Abbildung 8.1 dargestellt.

Das System läßt sich in zwei grundsätzlich verschiedene Teile untergliedern: Den Generator und das Laufzeitsystem. Der Generator liest die Spezifikation des Analysators und generiert daraus ANSI C Code. Der Code wird mit dem Laufzeitsystem zusammen übersetzt und bildet den Analysator. Dieser hat als Eingabe den Kontrollflußgraphen des zu analysierenden Programms und liefert als Ausgabe eine Annotierung desselben mit Datenflußinformationen.

8.1 Der Generator

Hier werden alle Eingabefiles zunächst mittels eines Scanners und eines Parsers zu einem abstrakten Syntaxbaum verarbeitet. Diese Teile wurden aus den Syntaxbeschreibungen der Sprachen mit `lex` und `yacc` generiert. Anschließend werden einige Konstrukte der Sprachen aufgelöst, indem man die entsprechenden Knoten durch andere einfachere Konstrukte ersetzt. Dazu zählt z.B. die '<=' Notierung bei **let**-Ausdrücken in FULA.

Anhand des Syntaxbaumes wird nun eine Typüberprüfung vorgenommen, die die Konsistenz der Spezifikation sicherstellt und jeden Knoten des Baumes mit dem zugehörigen Typ beschriftet. Anschließend wird aus den verschiedenen Teilen des Baumes Code generiert. Dazu gehört die Erzeugung der Datenstrukturen mit ihren abstrakten Interfaces, die Generierung der Transferfunktionen und die Erzeugung der Iterationsvorschrift.

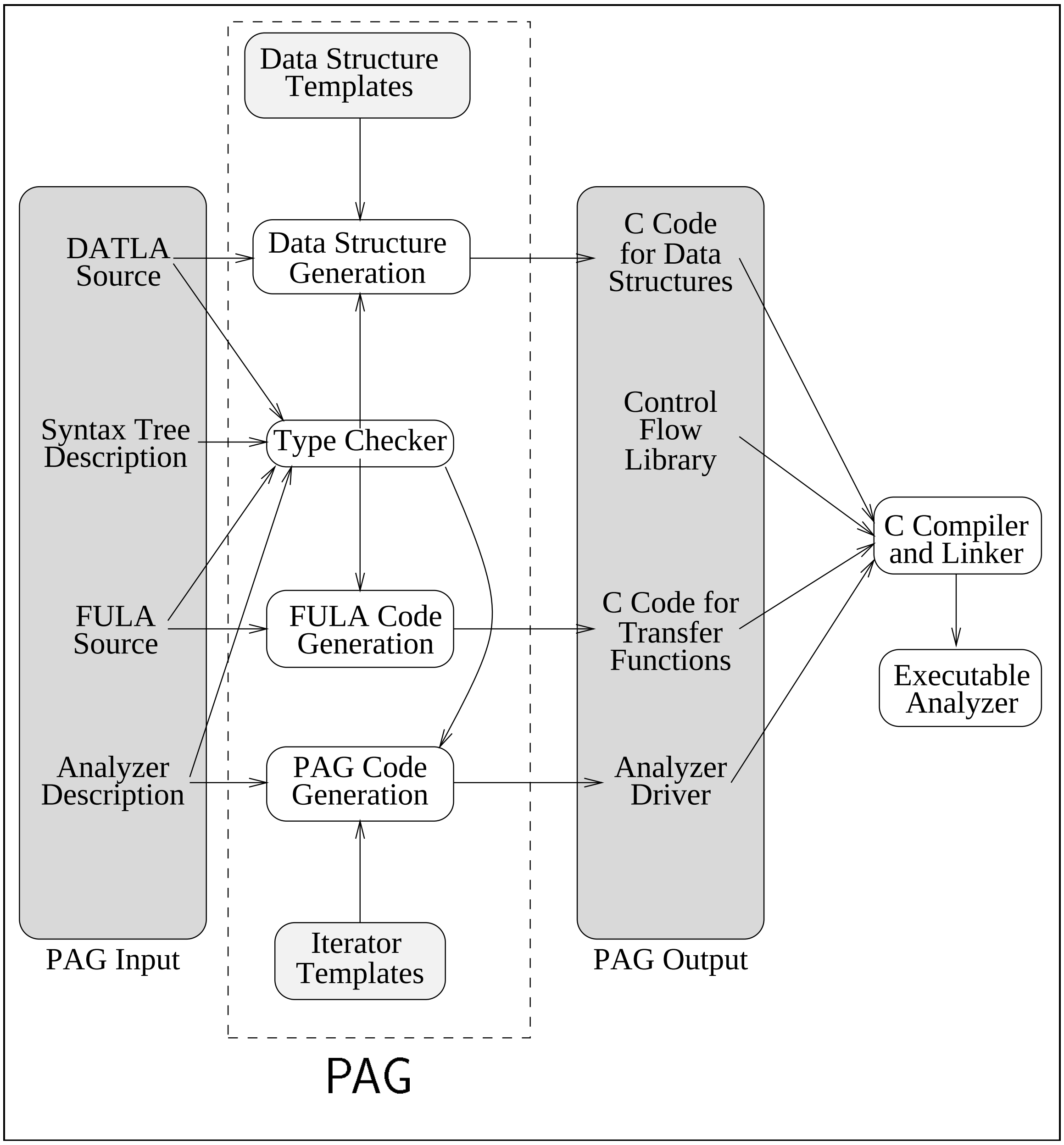

Abbildung 8.1: Die Struktur von PAG

8.1.1 Der Typanalysator

Der hier verwendete Algorithmus basiert auf einem Artikel von Cardelli [Car85]. Es ist ein Typinferenzalgorithmus für funktionale Sprachen. So kann für jeden Term von FULA ein Typ abgeleitet werden, der später zur Übersetzung nach C benötigt wird. Zur Bestimmung des Typs werden Typterme verwendet. Es gibt folgende Arten von Typen:

1. Typvariablen: Diese enthalten einen Zeiger auf einen anderen Typterm, falls sie instanziiert wurden. Falls dieser Zeiger ein Null Pointer ist, wurde die Variable bisher nicht instanziiert und steht für einen beliebigen Typ.

2. Basistypen: Diese stehen für Typen, die nicht weiter untergliedert sind, wie die vordefinierten Typen **unum**, **snum**, **real** , **char**, **string** oder Typen, die vom Syntaxbaum eingeführt werden.

3. Operatortypen: Sie setzen sich zusammen aus einem Typoperator und einer Liste von Argumenten, die wiederum Typterme sind. Folgende Operatoren werden benutzt:

   (a) Tupel: Ein Tupel von verschiedenen Einzeltypen. Die Anzahl der Argumente kann zwischen null und unendlich variieren.

   (b) Funktion: In FULA definierte Funktionen. Sie haben genau zwei (Typ-) Argumente, nämlich ein Tupel von Argumenttypen und einen Ergebnistyp.

   (c) Func: In DATLA definierte Funktionen. Sie haben ebenfalls zwei Argumente, allerdings muß das erste kein Tupel sein, da explizit mehrstellige Funktionen in DATLA nicht zulässig sind.

   (d) Set: Der Powertyp eines anderen Typen. Daher gibt es auch nur ein Argument: den Typ der Basiselemente.

   (e) List: Der Listentyp, ebenfalls mit einem Argument

   (f) Sum: Die disjunkte Summe mehrerer Typen. Diese werden als Argumenttypen benutzt.

Zur Unifikation zweier Typen muß dafür gesorgt werden, daß zuerst Ketten von instanziierten Variablen aufgelöst werden, d.h. ein Typterm wird in einen äquivalenten Typterm umgeformt, indem instanziierte Variablen dereferenziert werden. Ein solcher bereinigter Typterm kann nur entweder eine uninstanziierte Variable, ein Operatortyp oder ein Basistyp sein.

Dann ergibt sich die zur Unifikation nötige Aktion aus folgender Tafel:

$t_1 \backslash t_2$	Typvariable	Operator Typ	Basistyp
Typvariable	Instanziierung von t_1 auf t_2		
Operator Typ	Instanziierung t_2 auf t_1	siehe Text	fail
Basistyp	Instanziierung t_2 auf t_1	fail	unifizierbar falls $t_1 = t_2$

Bei der Instanziierung einer Variablen zu einem Typ muß zuerst ein Occur Check durchgeführt werden um sicherzustellen, daß die Variable nicht in dem Typterm vorkommt.

Zwei Operatortypen können nur dann erfolgreich unifiziert werden, wenn die Operatoren gleich sind und sich die Argumentlisten Element für Element unifizieren lassen.

8.1.2 Die Codegenerierung

Es gibt eine Codefunktion, um FULA Funktionen nach C zu übersetzen, zwei Funktionen, um Ausdrücke zu übersetzen und eine Funktion match, die den Code für Patternmatching erzeugt. Außerdem gibt es die Funktion func_tab, die einen Funktionsnamen in einen C-Funktionsnamen übersetzt. Diese Funktion ermöglicht es, z.B. eingebaute Funktionen in die korrekten C-Funktionen zu überführen.

Funktionen

Funktionen, die in FULA deklariert wurden, werden direkt in Funktionen in C übersetzt. Dies geschieht mittels der F_Code Funktion. Diese übersetzt eine Funktion f mit k Argumenten, die durch k Fälle definiert wurde.

$$
\mathsf{F\_Code}\left(\begin{array}{lcl} \mathrm{f}\left(\left\langle pat_{1,1}\right\rangle,\cdots,\left\langle pat_{1,k}\right\rangle\right) & = & \langle exp_1\rangle \\ & \vdots & \\ \mathrm{f}\left(\left\langle pat_{n,1}\right\rangle,\cdots,\left\langle pat_{n,k}\right\rangle\right) & = & \langle exp_n\rangle \end{array}\right) =
$$

$$
\left.\begin{array}{l} \mathsf{return\_type}(\mathrm{f})\ \ \mathsf{func\_tab}(\mathrm{f})\ \ (\ \ \mathsf{arg\_type}(\mathrm{f},1)\quad v_1, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{arg\_type}(\mathrm{f},k)\quad v_k \\ \qquad\qquad\qquad\qquad\qquad\quad) \end{array}\right\}\quad // \text{ C-Funktionskopf}
$$

$\{$

$$
\mathsf{match}([v_1,\cdots,v_k],\left[\begin{array}{c} \left(\left[\left\langle pat_{1,1}\right\rangle,\cdots,\left\langle pat_{1,k}\right\rangle\right],\langle exp_1\rangle\ \right), \\ \vdots \\ \left(\left[\left\langle pat_{n,1}\right\rangle,\cdots,\left\langle pat_{n,k}\right\rangle\right],\langle exp_n\rangle\ \right) \end{array}\right], v)
$$

fatal_error(„No pattern matched in f“); }
return(v);

$\}$

mit $v, v_1, \cdots, v_k$ neue Variablen

Ausdrücke

Ausdrücke können im allgemeinen nicht direkt nach C übersetzt werden, da z.B. in einem **let**-Ausdruck die Notwendigkeit besteht, Werte an Variablen zuzuweisen, was in C nur bedingt möglich ist. Daher wird die Übersetzung von FULA-Ausdrücken in zwei verschiedenen Kontexten vorgenommen, nämlich im E-Kontext, in dem direkt in C Ausdrücke (**E**xpressions) übersetzt wird, und im V-Kontext, in dem eine Folge von Statements erzeugt wird, die den Wert des Ausdrucks berechnet, und an eine Variable zuweist.

Es gibt zur Übersetzung von Ausdrücken zwei verschiedene Funktionen: Die Funktion E_Code, an die ein FULA-Ausdruck übergeben wird und die diesen in einen C-Ausdruck übersetzt, und die Funktion V_Code, die sowohl einen Ausdruck als auch eine Variable, an die der Wert zugewiesen werden soll, als Argumente hat und C-Statements produziert.

Es muß zunächst für den gesamten Quelltext festgelegt werden, ob die Ausdrücke im E– oder im V-Kontext übersetzt werden müssen. Dabei können alle Ausdrücke, die kein **let** oder **case**-Ausdruck sind oder einen Teilausdruck besitzen, der ein **let** oder ein **case**-Ausdruck ist, im E-Kontext übersetzt werden. Alle übrigen Ausdrücke müssen im V-Kontext übersetzt werden.

Nun sollen einige exemplarische Fälle der Code Funktionen, die über die Struktur des zu übersetzenden Ausdrucks definiert sind, aufgeführt werden:

$$\mathsf{E\_Code}\left(\textbf{if } \langle bool\_exp\rangle \begin{array}{ll}\textbf{then} & \langle exp_1\rangle \\ \textbf{else} & \langle exp_2\rangle\end{array} \textbf{ endif}\right) \quad = \quad \begin{array}{ll}\mathsf{E\_Code}(\langle bool\_exp\rangle) & ? \\ \mathsf{E\_Code}(\langle exp_1\rangle) & : \\ \mathsf{E\_Code}(\langle exp_2\rangle) & \end{array}$$

$$\mathsf{E\_Code}(\langle var\rangle) \quad = \quad \langle var\rangle$$

$$\mathsf{E\_Code}\left(\mathrm{f}\begin{pmatrix}\langle exp_1\rangle\,, \\ \vdots \\ \langle exp_n\rangle\end{pmatrix}\right) \quad = \quad \begin{array}{l}\mathsf{func\_tab}(\mathrm{f})(\\ \quad \mathsf{E\_Code}(\langle exp_1\rangle), \\ \qquad \vdots \\ \quad \mathsf{E\_Code}(\langle exp_n\rangle) \\ )\end{array}$$

falls f in FULA definiert wurde

$$\mathsf{E\_Code}(\mathrm{f}(\langle exp\rangle)) \quad = \quad \begin{array}{l}\mathsf{type}(\mathrm{f})\_\mathrm{get}(\\ \quad \mathrm{f}, \\ \quad \mathsf{E\_Code}(\langle exp\rangle) \\ )\end{array}$$

falls f in DATLA definiert wurde (es kann nur ein Argumente geben)

$$\mathsf{V\_Code}(\langle exp\rangle\,, v) \quad = \quad v = \mathsf{E\_Code}(\langle exp\rangle);$$

falls $\langle exp\rangle$ im E-Kontext übersetzbar ist

$$\mathsf{V\_Code}\left(\textbf{if } \langle bool\_exp\rangle \begin{array}{ll}\textbf{then} & \langle exp_1\rangle \\ \textbf{else} & \langle exp_2\rangle\end{array} \textbf{ endif}, v\right) \quad = \quad \begin{array}{l}\mathsf{V\_Code}(\langle bool\_exp\rangle\,, v_1); \\ \mathrm{if}\ (v_1)\ \{ \\ \quad \mathsf{V\_Code}(\langle exp_1\rangle\,, v); \\ \}\ \ \mathrm{else}\ \{ \\ \quad \mathsf{V\_Code}(\langle exp_2\rangle\,, v); \\ \}\end{array}$$

mit v_1 neue Variable

$$\mathsf{V\_Code}\left(\textbf{case}\begin{bmatrix} v_1, \\ \vdots \\ v_k \end{bmatrix}\textbf{of}\begin{bmatrix} \left\langle pat_{1,1}\right\rangle,\cdots,\left\langle pat_{1,k}\right\rangle & => & \langle exp_1\rangle\,; \\ & \vdots & \\ \left\langle pat_{n,1}\right\rangle,\cdots,\left\langle pat_{n,k}\right\rangle & => & \langle exp_n\rangle\,; \end{bmatrix}\textbf{endcase}, v\right)$$

$$= \quad \mathsf{match}\left(\begin{bmatrix} v_1, \\ \vdots \\ v_k \end{bmatrix}, \begin{bmatrix} \left(\left\langle pat_{1,1}\right\rangle,\cdots,\left\langle pat_{1,k}\right\rangle,\langle exp_1\rangle\right), \\ \vdots \\ \left(\left\langle pat_{n,1}\right\rangle,\cdots,\left\langle pat_{n,k}\right\rangle,\langle exp_n\rangle\right) \end{bmatrix}, v\right)$$

fatal_error(„No pattern matched in case expression“);

Patternmatching

Die Übersetzung des Patternmatching geschieht mittels einer abgewandelten Form des in [Jon87] angegebenen Algorithmus. Er basiert auf der Funktion match, die als Argumente eine Liste der zu matchenden Variablen, eine Liste von Paaren aus Listen von Pattern und einem Ausdruck und schließlich eine Variable erhält, in die das Funktionsergebnis geschrieben werden soll. Um ein exponentielles Wachstum des Codes durch Duplizieren von fail Code zu vermeiden, wird dieser mit Sprungmarken versehen und bei Bedarf angesprungen. Dazu muß die Codefunktion, die match aufruft, anschließend selbst den fail-Code erzeugen

und den Block durch eine schließende Klammer } beenden.

Es werden vier Fälle zur Definition der match-Funktion unterschieden. Im einfachsten Fall wurden durch rekursive Aufrufe der Funktion bereits alle Variablen gematcht und aus der Variablenliste entfernt. Da in diesem Fall kein Fehlschlag mehr erfolgen kann, kann der von der aufrufenden Code-Funktion erzeugtefail-Code niemals ausgeführt werden und also durch eine nachfolgende Optimierungsphase oder einen optimierenden C-Compiler entfernt werden.

$$\mathsf{match}\left([\,], \begin{bmatrix} ([\,],\langle exp_1\rangle), \\ \vdots \\ ([\,],\langle exp_n\rangle) \end{bmatrix}, v\right) \quad = \quad \begin{array}{l} \text{if (true) \{} \\ \quad \mathsf{V\_Code}\,(\langle exp_1\rangle\,, v) \\ \text{\} \ else \{} \end{array}$$

Im zweiten Fall beginnen alle Pattern mit einer Variablen. Dann kann in den zugehörigen Ausdrücken diese Variable durch die erste Variable der Variablenliste ersetzt werden. Anschließend werden die ersten Pattern sowie die erste Variable entfernt und match wieder aufgerufen.

$$\mathsf{match}\left(\begin{bmatrix} v_1, \\ \vdots \\ v_k \end{bmatrix}, \begin{bmatrix} (\langle var_1\rangle : r_1, \langle exp_1\rangle), \\ \vdots \\ (\langle var_n\rangle : r_n, \langle exp_n\rangle) \end{bmatrix}, v\right)$$

$$= \mathsf{match}\left(\begin{bmatrix} v_2, \\ \vdots \\ v_k \end{bmatrix}, \begin{bmatrix} (r_1, \langle exp_1\rangle [\langle var_1\rangle \mapsto v_1]) \\ \vdots \\ (r_n, \langle exp_n\rangle [\langle var_n\rangle \mapsto v_1]) \end{bmatrix}, v\right)$$

Im dritten Fall beginnen alle ersten Pattern der Patternliste mit einem Konstruktor. Dann können Listenelemente mit verschiedenen Konstruktoren umsortiert werden, ohne die Semantik zu verändern. Daher werden alle Elemente, deren erste Pattern mit dem gleichen Konstruktor beginnen, zusammengruppiert. Dann wird ein großes `case`-Statement (in C) sowie für jeden Fall ein neuer Aufruf von **match** erzeugt. Bei diesen Aufrufen werden nur die Patternlisten verwandt, die mit dem entsprechenden Konstruktor beginnen. Die Patternlisten und die Liste der zu matchenden Variablen werden jeweils um das erste Element verkürzt. Anschließend werden die Subpattern der gelöschten Pattern vor die jeweiligen Patternlisten gehängt und die Variablenliste um eine entsprechende Anzahl neuer Variablen erweitert. Da jedoch zum Test auf Gleichheit von Konstruktoren Funktionsaufrufe nötig sind, kann in C nur eine Kette von `if`-Abfragen statt eines `case`-Statements benutzt werden.

$$\mathsf{match}\left(\begin{bmatrix} v_1, \\ \vdots \\ v_e \end{bmatrix}, k_1 \mathbin{++} \cdots \mathbin{++} k_n, v\right)$$

$$\text{mit } k_i = \begin{bmatrix} \left(\langle cons_i\rangle \begin{pmatrix} \left\langle pat_{i,1,1}\right\rangle, \\ \vdots \\ \left\langle pat_{i,1,s_i}\right\rangle \end{pmatrix} : rs_{i,1}, \left\langle exp_{i,1}\right\rangle\right), \\ \vdots \\ \left(\langle cons_i\rangle \begin{pmatrix} \left\langle pat_{i,j_i,1}\right\rangle, \\ \vdots \\ \left\langle pat_{i,j_i,s_i}\right\rangle \end{pmatrix} : rs_{i,j_i}, \left\langle exp_{i,j_i}\right\rangle\right) \end{bmatrix}$$

$$\left.\begin{array}{l}
= \text{if (equal(constructor}(v_1), \langle cons_i \rangle))\,\{ \\
\quad \begin{array}{lcl} w_1 & = & \text{subexpression}(v_1, 1); \\ & \vdots & \\ w_{s_i} & = & \text{subexpression}(v_1, s_i); \end{array} \\
\quad \mathsf{match}\left(\begin{bmatrix} w_1, \\ \vdots \\ w_{s_i}, \\ v_2, \\ \vdots \\ v_e \end{bmatrix}, \begin{bmatrix} \left(\begin{bmatrix} \left\langle pat_{i,1,1} \right\rangle, \\ \vdots \\ \left\langle pat_{i,1,s_i} \right\rangle \end{bmatrix} +\!\!+\ rs_{i,1}, \left\langle exp_{i,1} \right\rangle \right), \\ \vdots \\ \left(\begin{bmatrix} \left\langle pat_{i,j_i,1} \right\rangle, \\ \vdots \\ \left\langle pat_{i,j_i,s_i} \right\rangle \end{bmatrix} +\!\!+\ rs_{i,j_i}, \left\langle exp_{i,j_i} \right\rangle \right) \end{bmatrix}, v \right) \\
\quad \text{goto } l; \} \\
\} \text{ else } \{ \\
l:
\end{array}\right\} \begin{array}{c} \text{für} \\ 1 \le i \le n \end{array}$$

mit w_i neue Variablen, l neue Sprungmarke

Im letzten Fall trifft keine der Bedingungen der ersten drei Fälle zu. d.h. ws beginnen einige erste Pattern mit Variablen, andere mit Konstruktoren. Dann kann der Aufruf in mehrere Aufrufe der **match** Funktion aufgespalten werden, bei denen jeweils nur Variablen oder nur Konstruktoren an erster Stelle stehen. Die nachfolgenden match Aufrufe werden dabei als **fail**-Code der vorangegangenen verwendet.

$$\mathsf{match}\begin{pmatrix} v_1, & \\ \vdots & , k_1 +\!\!+ \cdots +\!\!+ k_n, v \\ v_e & \end{pmatrix} \quad = \quad \begin{array}{l} \mathsf{match}\begin{pmatrix} v_1, & \\ \vdots & , k_1, v \\ v_e & \end{pmatrix} \\ \quad\quad \vdots \\ \quad \mathsf{match}\begin{pmatrix} v_1, & \\ \vdots & , k_n, v \\ v_e & \end{pmatrix} \\ \quad\quad \mathsf{fail}\text{-Code} \\ \\ \} \cdots \} \qquad\qquad\qquad // \ n \text{ mal} \end{array}$$

mit k_i sind jeweils maximal, so daß die ersten Pattern nur mit einer Variablen oder nur mit einem Konstruktor beginnen.

8.2 Das Laufzeitsystem

Das Laufzeitsystem besteht aus einigen Teilen, die unabhängig vom zu implementierenden Analysator sind, also nur zum System hinzugelinkt werden müssen. Dies sind z.B. die Teile zur Verwaltung der Worklist oder des Kontrollflußgraphen. Andere müssen mit Teilen der Spezifikation parametrisiert werden, wie die Verwaltung der Datenflußinformationen, die u.a. den Datentyp der Flußelemente benötigt, oder der Iterationsalgorithmus. Diese liegen als Templatefiles vor, in denen Metavariablen durch aktuelle Werte ersetzt werden müssen. Ein letzter Teil des Codes muß vollständig generiert werden, da die Anzahl der benötigten Parameter von der Problemstellung abhängt, wie bei dem Datentyp der Tupel.

Für alle Module des Laufzeitsystems gibt es ein genau festgelegtes Interface, so daß es leicht möglich ist, diese gegen andere Implementierungen auszutauschen.

8.2.1 Iteratoren

Die Iterationsalgorithmen liegen als Templatefiles vor und sind damit einfach zu programmieren. Bisher wurde ein Worklist Algorithmus mit chaotischer Iterationsfolge implementiert. Er entspricht dem Pseudocode in Abbildung 4.2 auf Seite 34.

8.2.2 Datentypen

Jeder in DATLA definierte oder eingebaute Datentyp wird durch ein getrenntes C-File und eine C-Datenstruktur implementiert. Daher ist es wichtig, ein genau definiertes Interface an Grundfunktionen zur Verfügung zu stellen. Hinzu kommen noch einige Datentyp spezifische Funktionen, wie Projektionen für Tupel. Der Aufbau eines Datentyps soll nun anhand der Implementierung für Mengen und Funktionen, die besonders interessant sind, aufgezeigt werden.

Mengen

Eine Menge T kann auf zwei Arten repräsentiert werden. Die naheliegendste ist die Repräsentierung durch eine Liste, die alle Elemente der Menge enthält. Dieses Verfahren erlaubt es aber nicht, falls T unendlich viele Elemente hat, bei einer Datenflußanalyse mit $\top$ (also der Menge aller Elemente von T) zu beginnen und iterativ Elemente zu entfernen. Man kann eine Menge auch subtraktiv darstellen, d.h. die Elemente der Liste sind gerade die Elemente von T, die nicht in der Menge enthalten sind. Es wurde hier eine kombinierte Darstellung gewählt, die aus einem

Flag besteht, um subtraktive oder additive Darstellung zu unterscheiden, und einer Liste von Elementen aus T.

Falls also z.B. das Flag eine subtraktive Mengendarstellung angibt und die Liste aus einem Element e besteht, so wird die Menge aller Elemente von T ohne e dargestellt. Auf diese Weise kann die Menge aller Elemente, auch eines unendlichen Typs, dargestellt werden.

Da nicht alle Mengen über einen Typ T auf diese Weise dargestellt werden können (z.B. nicht die Menge aller geraden Zahlen), ist es notwendig sicherzustellen, daß keine der benötigten Funktionen die Klasse der darstellbaren Mengen verläßt. Für das Hinzufügen und Entfernen einzelner Elemente ist dies offensichtlich. Für die Operationen Vereinigung und Schnitt wird es aus den Algorithmen ersichtlich, die weiter unten angegeben sind.

Vorsicht ist auch bei endlichen Typen T geboten, da falls z.B. $T = \{a, b\}$ ist, die Negativdarstellung mit der liste $[a, b]$ bereits die leere Menge ist. Daher muß bei der Überprüfung auf leere Menge, was äquivalent zum Test auf $\perp$ ist, die Mächtigkeit des Typs T bekannt sein. Diese ist bei jedem Typ durch die Funktion T_power zu erfahren.

Bei der Vereinigungs und Schnitt Operation werden drei verschiedene Operationen auf Listen benötigt. Dabei wird vorausgesetzt, daß in den Listen keine Elemente doppelt vorkommen. Alle Operationen sind binär, und ihre Argumente werden im folgenden L_1 und L_2 genannt.

1. Listen-Vereinigung $\bigcup$: Alle Elemente aus Liste L_2 werden zu Liste L_1 hinzugefügt und anschließend doppelte Vorkommen entfernt.

2. Listen-Schnitt $\bigcap$: Das Ergebnis ist eine Liste, die nur die Elemente enthält, die sowohl in L_1 als auch in L_2 vorhanden sind.

3. Listen-Restbildung $\biguplus$: Bestimmt alle Elemente, die in L_1 und nicht in L_2 enthalten sind. Diese Operation ist im Gegensatz zu den zwei vorgenannten nicht kommutativ.

Nun folgen zwei Tabellen, eine für die Vereinigung auf Mengen, die andere für den Schnitt. Dabei ergeben sich je vier Fälle, je nachdem ob M_1 und M_2 additiv ($\oplus$) oder subtraktiv ($\ominus$) dargestellt sind. Das Ergebnis der Operation ist eine Menge, deren Liste sich durch die in der Tabelle angegebene Operation auf den zu M_1 und M_2 gehörigen Listen L_1 bzw. L_2 ergibt. Das Flag der resultierenden Menge ist ebenfalls in der Tabelle angegeben. Falls also z.B. der Schnitt von M_1 und M_2 berechnet werden soll und M_1 additiv und M_2 subtraktiv gegeben ist, so ist das Ergebnis eine Menge in additiver Darstellung, deren Liste sich aus der Restbildung von L_2 und L_1 ergibt.

Vereinigung:

		M_2	
		$\oplus$	$\ominus$
M_1	$\oplus$	$L_1 \bigcup L_2$ $\oplus$	$L_2 \biguplus L_1$ $\ominus$
	$\ominus$	$L_1 \biguplus L_2$ $\ominus$	$L_1 \bigcap L_2$ $\ominus$

Schnitt:

		M_2	
		$\oplus$	$\ominus$
M_1	$\oplus$	$L_1 \bigcap L_2$ $\oplus$	$L_2 \biguplus L_1$ $\oplus$
	$\ominus$	$L_2 \biguplus L_1$ $\oplus$	$L_1 \bigcup L_2$ $\ominus$

Nun ist klar, daß die Klasse der darstellbaren Mengen unter den zugelassenen Operationen Vereinigung, Schnitt, „Element hinzufügen“ und „Element entfernen“ abgeschlossen ist.

Funktionen

Funktionen $T_1 \rightarrow T_2$ werden durch ein default Element aus T_2 und eine doppelt verkettete Liste von Paaren von Feldern von T_1 und T_2 dargestellt. Dies ist ein hybrides Verfahren aus der Darstellung als Liste von Paaren, die ineffizient ist, und der Darstellung als Paar von Feldern, die eine statische Größe haben und somit nicht beliebige Funktionen darstellen können.
Mittels des Parameters der Feldgröße kann die Implementierung zu beiden Extremen hin verändert werden. Die Datenstruktur wird in Abbildung 8.2 verdeutlicht.

Bei der Darstellung gelten folgende Invarianten, um die Konsistenz zu garantieren:

1. Kein Element in den Listen wird auf den default Wert abgebildet

2. Es gibt keine leeren Blöcke

3. Kein Element kommt doppelt in den Listen vor

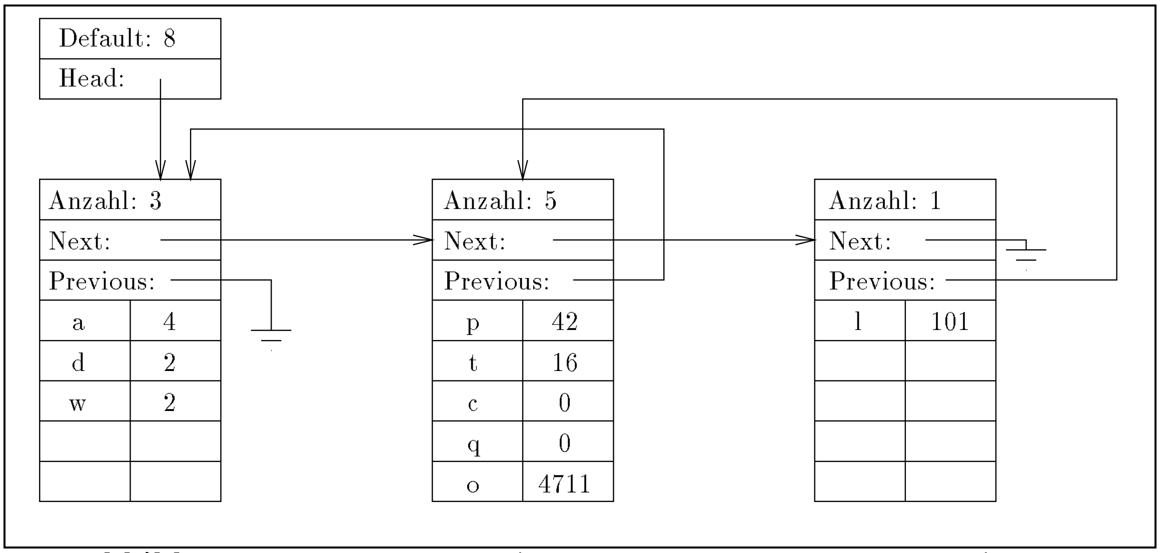

Abbildung 8.2: Die Datenstruktur zur Repräsentierung von Funktionen

Auch hier gilt ebenso wie bei den Mengen, daß keine Operation es ermöglicht, die Menge der darstellbaren Funktionen zu verlassen. Nicht darstellbar ist z.B. die Funktion

$$f(x) = \begin{cases} 1 & \text{falls } x \text{ gerade} \\ 0 & \text{sonst} \end{cases}$$

8.2.3 Garbage Collector

Da in FULA Objekte durch Pointer dargestellt werden und mehrere Pointer auf ein Objekt zeigen können, ist es nötig, eine Copy Semantik einzuführen. Wenn also eine Funktion ein Objekt verändert, muß sie es vorher kopieren und das ursprüngliche Objekt unverändert lassen. Deshalb ist nicht statisch entscheidbar, wann ein Objekt nicht mehr gebraucht wird und sein Speicherplatz freigegeben werden kann. Daher wurde für FULA ein Garbage Collector implementiert, um den Platzverbrauch zu minimieren.

Der Collector arbeitet nach dem Mark and Scan Prinzip, in dem zu einem bestimmten Zeitpunkt alle noch benötigten Objekte und die von diesen erreichbaren Objekte markiert werden. Anschließend werden die noch benötigten Objekte zusammengeschoben und alle Zeiger aktualisiert.

Es ist also nötig, alle zu einem bestimmten Zeitpunkt noch benötigten Objekte zu kennen. Dies sind bei einem iterativen Algorithmus zwischen den Neuberechnungen von Datenflußelementen nur alle aus dem Datenflußgraphen heraus erreichbaren Objekte. Zusätzlich können noch einige statische Objekte existieren, die ebenfalls markiert werden müssen.

Daher wurde folgendes Vorgehen gewählt:

1. Jeder Datentyp hat eine *mark* Funktion, um alle in einem Objekt enthaltenen Pointer zu markieren. Falls es sich bei dem Objekt selbst nicht um einen Pointer handelt, sondern um einen Wert, so kann dies hier behandelt werden.

2. Jeder Datentyp hat eine Funktion, um seine statischen Objekte zu markieren. Statisch werden z.B. bei den meisten Typen $\top$ und $\perp$ gehalten, um auf diese schnell zugreifen zu können.

3. Speicherallokierung und Freigaben müssen über die Speicherverwaltung des Garbage Collectors abgewickelt werden.

Durch dieses festgelegte und universelle Interface ist es möglich, verschiedene Implementierungen für die Speicherverwaltung und den Collector zu wählen. Selbstverständlich ist es, um Rechenzeit zu sparen, auch möglich, den Garbage Collector auszuschalten, falls genügend Speicher vorhanden ist oder falls nur kleine Programme analysiert werden sollen.

Kapitel 9

Praktische Messungen

PAG generierte Analysatoren wurden erfolgreich sowohl in einem großen Compiler System mit ANSI C und Fortran90 Frontends als auch mit verschiedenen Backends (einschließlich SPARC) im **ESPRIT** Projekt #5399 (COMPARE) getestet. Folgende Datenflußanalysen wurden spezifiziert:

- Copy-Konstantenpropagation: Nur Zuweisungen der Form `x := c` wobei `c` eine Konstante ist und `x := y` werden berücksichtigt. Hier ist die Spezifikation besonders einfach, da kein vollständiger Interpreter für Ausdrücke der Programmiersprache benötigt wird.

- Lineare Konstantenpropagation: Hier werden zusätzlich Anweisungen der Form `x := c * y + d` betrachtet, wobei `c` und `d` Konstanten sind. Der Vorteil gegenüber der vollen Konstantenpropagation besteht darin, daß der verwendete Verband endlich und das Problem distributiv ist.

- Volle Konstantenpropagation: Jede rechte Seite wird berücksichtigt. Der Verband einer solchen Analyse ist unendlich groß, außerdem sind die Transferfunktionen nicht distributiv (siehe Bild 9.1). Dies verlangt das allgemeinste Analyseverfahren, um auch hier eine Terminierung garantieren zu können.

- Alias Analyse: Berechnet *must* und *may* Aliase

- Liveness: Ein Bitvektorproblem, bei dem für jeden Knoten die Menge der lebendigen Variablen berechnet wird. Bislang existiert keine spezielle Implementierung für Bitvektoren. Vielmehr werden diese durch Funktionen simuliert. Ein Prototyp für eine solche Datenstruktur wird zur Zeit getestet. Damit kann bei geeigneten Problemen Platz und Zeit gespart werden.

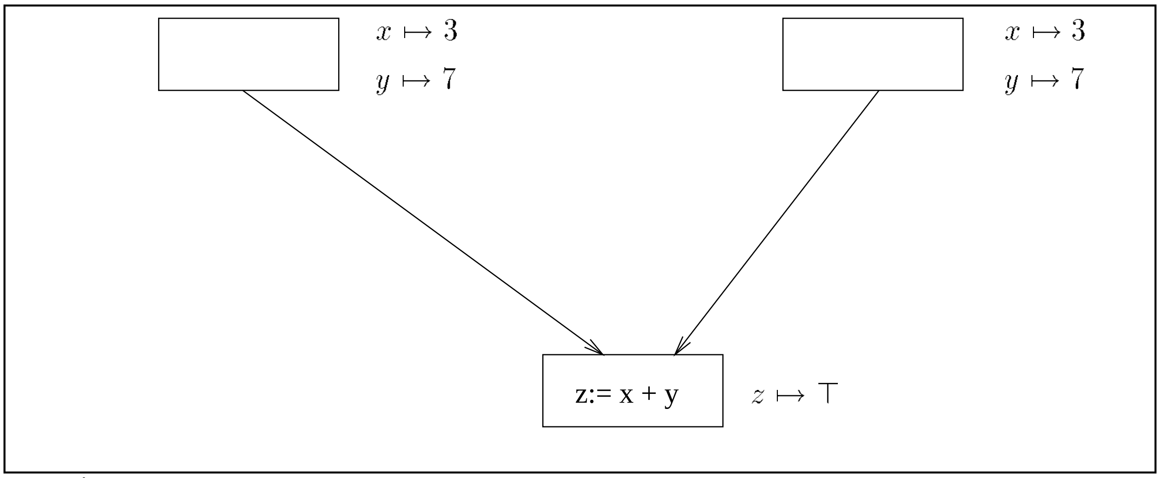

Abbildung 9.1: Dieses Bild zeigt, daß der Verband zur Konstantenpropagation nicht distributiv ist.

Analyse	Spezifikationszeilen				Generiertes	Interface
	DATLA	FULA	Syntax	$\sum$	(kB)	(Zeilen)
copy constant propagation	233	12	144	389	268	1283
linear constant propagation	400	14	144	558	309	1251
full constant propagation	628	12	144	784	371	1283
alias	203	10	144	357	215	1084
liveness	95	10	144	249	222	1054
dominator	22	3	144	169	147	1084

Abbildung 9.2: Die Größe der Spezifikationen und der generierten Analysatoren

- Dominator: Berechnet zu jedem Knoten die Dominatoren, was sich ebenfalls als Bitvektorproblem darstellen läßt.

In Abbbildung 9.2 ist die Größe der verschiedenen Spezifikationen und Analysatoren tabelliert. Die Spezifikation des Syntaxbaumes ist für alle Analysatoren gleich, ebenso wie ein großer Teil des handcodierten Interfaces, das in der letzten Spalte aufgeführt ist. Da es einige Unterschiede in der nachfolgenden Behandlung der berechneten Daten gibt, sind diese nicht vollständig identisch.

Es sollte die praktische Relevanz des universellen Mappings aus Kapitel 5.2.4 auf Seite 59 getestet werden. Dazu wurde die volle Konstantenpropagation und die Copy-Konstantenpropagation auf einige ‚reale' Programme angewandt. Die Ergebnisse sind in Anhang B (Seite 111) zu sehen. Dies soll nur den Auftakt zu weiteren Messungen in Zukunft darstellen. Es hat sich aber bereits gezeigt, daß PAG generierte Analysatoren mit großen und sogar sehr großen Programmen zurechtkommen. Allerdings stellte sich heraus, daß die Anforderungen an die Rechenzeit und den Speicherplatz mitunter sehr groß werden können. Daher sollen in Zukunft andere Fixpunktalgorithmen und Mappings zur Beschleunigung der Konvergenz implementiert werden. Außerdem soll die Speicherverwaltung effizienter gestaltet werden.

Es wurden auch Analysatoren für ein anderes Compilersystem generiert, nämlich für **CLAX** (siehe [COM90]). CLAX ist eine Pascal ähnliche imperative Sprache, die allerdings auf Zeiger verzichtet und so Analysen vereinfacht, da Aliase nur durch Parameterübergaben erzeugt werden können.

Kapitel 10

Verwandte Arbeiten und Ausblicke

10.1 Andere Arbeiten

Das Gebiet der Generierung von Datenflußanalysatoren ist nicht neu. Schon in [Hec77] wird ein allgemeines Framework zu Spezifikation von Datenflußproblemen angegeben.

So wurden auch in vielen Papieren Analysator-Generatoren vom theoretischen Standpunkt aus diskutiert. Allerdings wurden nicht sehr viele ernsthafte Versuche unternommen, diese auch praktisch zu implementieren. Einige davon sollen nun beschrieben werden.

10.1.1 Spare

Dieses System wurde in [VF92] von Venkatesch und Fischer vorgestellt. Es wurde entworfen, um automatische Implementierungen für Testzwecke zu erhalten. Es basiert direkt auf der abstrakten Interpretation. Der Benutzer kann eine abstrakte denotationale Semantik der zu analysierenden Zwischensprache angeben und ebenfalls die benötigten Domains spezifizieren. Entsprechend existiert keine explizite Unterstützung der intraprozeduralen Analyse.

Der gravierende Unterschied zu PAG besteht darin, daß Spare nie dazu entworfen wurde, Analysatoren für Übersetzungssysteme zu generieren. Vielmehr erzeugt es Code für den Synthesizer Generator, dessen Ausgabe ein Editor ist, der auch die spezifizierte Analyse ausführen kann. Dieser kann dann dazu verwendet werden, die Analyse zu testen, während PAG generierte Analysatoren bewiesen haben, daß sie für den Einsatz in fast beliebigen kompilierenden Systemen geeignet sind.

10.1.2 Sharlit

Tjiang und Hennessy haben in [TH92] ihr Tool Sharlit vorgestellt. Es basiert wie PAG auf Datenflußanalyserahmen. Allerdings werden hier weder die benötigten Transferfunktionen noch der Code für die zugrunde liegenden Verbände generiert, sondern müssen direkt vom Benutzer in C++ implementiert werden. Daher beschränkt sich die Arbeit des Tools auf die Generierung eines Iterationsalgorithmus. Dieser basiert auf Pfadkomprimierungen. Dazu wird die Menge aller Pfade eines Startknotens zu einem Endknoten durch einen Pfadausdruck beschrieben, der die Knoten mittels Konkatenation, Alternativenbildung und der Hüllenbildung verbindet. Für solche Pfadausdrücke kann dann eine neue Transferfunktion aus den Transferfunktionen der Knoten berechnet werden, falls der Benutzer die entsprechende Funktionen implementiert hat. Die Klasse der Probleme, auf die dieses Verfahren angewandt werden kann, ist aber eingeschränkt, da der zugrunde liegende Verband Operationen, wie den meet auf Funktionen (!), zulassen muß. Andererseits muß, wie bei der Intervallanalyse, der Flußgraph reduzierbar sein.

Sharlit umgeht dieses Problem, indem es nur versucht, Teilgraphen zu komprimieren, die mit den implementierten Operationen komprimierbar sind. So werden z.B. gerade die basic Blocks komprimiert, falls nur die Konkatenation von Funktionen implementiert ist. Auf dem verbleibenden Graphen wird dann eine Standarditeration durchgeführt.

Fazit

Sharlit benutzt einen interessanten hybriden Ansatz zwischen Pfadkomprimierung und Standarditeration, um die Fixpunktberechnung zu beschleunigen, falls es der Verband zuläßt. Allerdings hat der Benutzer weite Teile der Analyse nach wie vor in C zu implementieren mit allen bekannten Problemen.

10.1.3 System-Z

System Z ist ebenfalls ein Programm zur Erzeugung von Programmanalysatoren. Es wurde in [YH93] von Yi und Harrison ausführlich beschrieben.

Basis bildet hier die abstrakte Interpretation, so daß der Benutzer einen abstrakten Interpreter in einer funktionalen Sprache definieren kann. Auch die benötigten Datenstrukturen werden automatisch erzeugt. Dabei können diese, im Gegensatz zu PAG, nicht rekursiv definiert werden. Trotzdem ist es dem Benutzer überlassen, die Terminierung der Analyse zu garantieren.

System Z hat eine interessante Möglichkeit, um zwischen Genauigkeit der Analyse und ihren Kosten zu variieren. Dazu werden sogenannte Projektionen verwendet, mit denen im nachhinein die Analyse beschleunigt werden kann. Durch eine Projektion werden bestimmte Elemente eines Verbandes auf $\top$ abgebildet, so daß aufsteigende Ketten verkürzt werden können, aber auch die entsprechenden Informationen verloren gehen.
Diese Methode ist vergleichbar mit einer eingeschränkten Form des Widening aus Kapitel 4.2.4.

10.1.4 Sonstige

Steffen und Knoop beschreiben in verschiedenen Papieren ([Kno93, KRS95]) ein Werkzeug zur Generierung von Datenflußanalysatoren. Dieses wurde aber meines Wissens nie implementiert, so daß es sich um eine weitere Beschreibung der Datenflußanalyserahmen handelt.

Auch der geistige Vater der abstrakten Interpretation, Cousot, hat bis heute kein Werkzeug vorgestellt, mit dem sich auf einfache Art abstrakte Analysen, auch nur zu Testzwecken, implementieren lassen.

10.2 Fazit und Ausblick

Es wurde gezeigt, daß PAG in der Lage ist, Analysatoren zu generieren, die auch mit sehr großen Programmen zurechtkommen. Um eine komplexe Analyse wie die Copy-Konstantenpropagation für ANSI C zu spezifizieren, benötigt man maximal vier Stunden, wenn das Interface bereits existiert. (Dieses wird aber auch nur einmal für jedes kompilierende System benötigt, für das mit PAG Analysen generiert werden sollen.) So war es auch möglich, eine existierende Spezifikation für die Konstantenpropagation in weniger als drei Stunden um eine einfache Aliasanalyse zu erweitern. Für die Zukunft sind folgende Erweiterungen geplant:

- Neue Fixpunktalgorithmen:
  Es sollen neue Fixpunktalgorithmen implementiert werden. Außerdem soll auch untersucht werden, ob es möglich ist, einen hybriden Ansatz zu verwenden, in dem man verschiedene Algorithmen mischt, und trotzdem die Korrektheit und die Terminierung garantieren kann.

- Reduzierung des Platz- und Zeitverbrauchs:
  Die Vektoren von Datenflußelementen können sehr groß sein, falls das Programm hauptsächlich aus Prozeduraufrufen besteht. In diesem Fall sind aber die Informationen an den meisten Prozedurköpfen gleich, so daß sie zusammengefaßt werden können. Dies hätte zwar eine Verschlechterung des Ergebnisses zur Folge, aber reduziert den Zeit- und Platzverbrauch. Gut wäre es, diese Zusammenfassung während der Analyse dynamisch vorzunehmen. Dazu könnte der Benutzer durch strukturelle Induktion über den Verband eine Distanz zweier Elemente definieren, so daß Elemente mit geringerer Distanz, falls nötig, zusammengefaßt werden können. Die $\mathbf{map}_c$ Funktionen müssen dann ebenfalls dynamisch angepaßt werden.

- Automatische Beweise über die Spezifikation:
  Eine interessante Forschungsrichtung könnte die Analyse der PAG Spezifikation sein. Die first order Sprache FULA mit einer eager Semantik erlaubt es automatisch, Eigenschaften der Spezifikation, wie die Monotonie oder Distributivität, zu beweisen und so die Terminierung zu garantieren bzw. einen anderen Fixpunktalgorithmus zu wählen.

Anhang A

Spezifikation der Konstantenpropagation

```
SYNTAX

START : mirSTMT

mirSTMT:
     mirEvaluate(Expr:mirEXPR)
   | mirBeginProcedure(Params:mirObject*,Locals:mirObject*)
   | mirAssign(Lhs:mirEXPR,Rhs:mirEXPR)
   | mirCall(Proc:mirEXPR,Params:mirEXPR*)
   | mirFuncCall(Proc:mirEXPR,Params:mirEXPR*,Res:mirEXPR)
   | mirCallRet(Params:mirObject*,Locals:mirObject*)
   | mirHandler(IdObj:mirEXPR,ParamObj:mirEXPR)
   | mirPhshHandler(Handler: mirEXPR)
   | mirPopHandler()
   | mirHeapAllocate(Type:mirTYPE,Size:mirEXPR,Res:mirEXPR)
   | mirStackAllocate(Type:mirTYPE,Size:mirEXPR,Res:mirEXPR)
   | mirEndProcedure(Params:mirObject*,Locals:mirObject*)
   | mirGoto(Target:mirEXPR)
   | mirReturn(Value:mirEXPR,Next:mirEXPR)
   | mirTailCall(Proc:mirEXPR,Params:mirEXPR*)
   | mirIf(Cond:mirEXPR,Then:mirEXPR,Else:mirEXPR)
   | mirSwitch(Value: mirEXPR, Cases: mirCase*)
   | mirRaise(Id:mirEXPR,Param:mirEXPR)
   | mirEndFuncCall(Res:mirEXPR)
   | mirEndCall()
   | mirExternProc(Name:NAME)
   | mirUnknownProc(Name:NAME);

mirCase:
     mirCase(Lwb:mirEXPR,Upb:mirEXPR);

mirTYPE:
     mirNoType()
   | mirAnyType()
   | mirBoolean()
   | mirBasicBlockType()
   | mirInteger(Signed:BOOL)
   | mirReal()
   | mirPointer(RefType:mirTYPE,Voilatile:BOOL)
```

```
   | mirArray(ElemType:mirTYPE,Lwb:mirEXPR,Upb:mirEXPR)
   | mirRecord(Fields:mirObject*)
   | mirUnion(Members:mirObject*)
   | mirProcType(ResultType:mirTYPE);

mirEXPR:
     mirNoExpr(Type:mirTYPE)
   | mirIntConst(Type:mirTYPE,Value:UNIV_INT)
   | mirRealConst(Type:mirTYPE,Value:UNIV_REAL)
   | mirBoolConst(Type:mirTYPE,Value:BOOL)
   | mirAddrConst(Type:mirTYPE,Value:UNIV_ADDRESS)
   | mirArrayCom(Type:mirTYPE,Value: mirEXPR*)
   | mirRecordCom(Type:mirTYPE,Value:mirEXPR*)
   | mirUnionCom(Type:mirTYPE,Value:mirEXPR,Part:mirObject*)
   | mirTypeSize(Type:mirTYPE,OfType:mirTYPE)
   | mirObjectAddr(Type:mirTYPE,Obj:mirObject)
   | mirContent(Type:mirTYPE,Addr:mirEXPR)
   | mirSubscript(Type:mirTYPE,Base:mirEXPR,Index:mirEXPR)
   | mirMember(Type:mirTYPE,Base:mirEXPR,Field:mirObject*)
   | mirBitExtract(Type:mirTYPE,Src: mirEXPR,Start:mirEXPR,Width:mirEXPR)
   | mirBitInsert(Type:mirTYPE,Dest: mirEXPR,Start:mirEXPR,Width:mirEXPR,Src:mirEXPR)
   | mirBoolVal(Type:mirTYPE,Value:mirEXPR,True:UNIV_INT,False:UNIV_INT)
   | mirConvert(Type:mirTYPE,Value:mirEXPR,Rounding:mirROUNDING)
   | mirCast(Type:mirTYPE,Value:mirEXPR)
   | mirCheckReal(Type:mirTYPE,Value:mirEXPR)
   | mirCheckRange(Type:mirTYPE,Value:mirEXPR)
   | mirAbs(Type:mirTYPE,Value:mirEXPR)
   | mirNeg(Type:mirTYPE,Value:mirEXPR)
   | mirNot(Type:mirTYPE,Value:mirEXPR)
   | mirAddrPlus(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirAddrDiff(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirCompare(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL,Relation:mirRELATION)
   | mirPlus(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirDiff(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirMult(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirShiftLeft(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirShiftRight(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirShiftRightSign(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirAnd(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirOr(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirXor(Type:mirTYPE,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirDiv(Type:mirTYPE,OnZero:mirEXPR,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirQuo(Type:mirTYPE,OnZero:mirEXPR,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirMod(Type:mirTYPE,OnZero:mirEXPR,Left:mirEXPR,Right:mirEXPR,Strict:BOOL)
   | mirRem(Type:mirTYPE,OnZero:mirEXPR,Left:mirEXPR,Right:mirEXPR,Strict:BOOL);

mirRELATION:
     Less()
   | LessEqual()
   | Greater()
   | GreaterEqual()
   | Equal()
   | UnEqual();


mirROUNDING:
     Truncation()
   | Nearest()
   | Floor()
   | Ceiling();

mirObject:
```

```
     mirDataGlobal(cpid:INT,Type:mirTYPE,IsVolatile:BOOL
       ,Name: NAME,Section: mirSection)
   | mirProcGlobal(cpid:INT,Type:mirTYPE,IsVolatile:BOOL
       ,Name: NAME,Section: mirSection)
   | mirBasicBlock(cpid:INT,Type:mirTYPE,IsVolatile:BOOL
       ,Name: NAME,Section: mirSection)
   | mirLocal(cpid:INT,Type:mirTYPE,IsVolatile:BOOL
       ,Procedure:mirObject)
   | mirParameter(cpid:INT,Type:mirTYPE,IsVolatile:BOOL,Procedure:mirTYPE
       ,ParamKind:mirParamKIND)
   | mirField(cpid:INT,Type:mirTYPE,IsVolatile:BOOL ,Composite:mirTYPE)
   | mirPlaced(cpid:INT,Type:mirTYPE,IsVolatile:BOOL ,Address:UNIV_ADDRESS)
   | mirRegister(cpid:INT,Type:mirTYPE,IsVolatile:BOOL ,RegisterId:INT)
   | ExternProc(Name:NAME)
   | UnknownProc(Name:NAME);


mirSection:
     mirSection(Sname:NAME);

mirParamKIND:
     ByValue()
   | ByReference()
   | ByCopyInOut();

BOOL == bool;

NAME==str;

INT==snum;

UNIV_INT == snum;

UNIV_REAL == real;

UNIV_ADDRESS == unum;




SET

noth_dom = (nix | | nix | nix )
const_val = bool ++ snum ++ real ++ noth_dom ++ unum
numbool = unum * bool

DOMAIN
va = flat(const_val)
const_fkt = unum -> va
dfi   = const_fkt * va * const_fkt
// Ein Dreitupel aus: normalem Datenflusswert, Rueckgabewert von
// Prozeduren und Parameter




PROBLEM Constant
direction : forward
carrier   : dfi
init     : bot
combine   : comb
init_start: ([->top],top,[->top])
```

```
TRANSFER

mirAssign(mirObjectAddr(_,obj),e2):
   ( if is_volatile(obj) then @!1!3 else @!1!3\[id(obj) -> value(e2,@!1!3)] endif
     , top, [->top])

// Zuweisungen an Pointer
mirAssign(mirContent(_,_),_):
   ( [->top],top,[->top])

mirUnknownProc(_):
   ( [->top],top,[->top])


mirFuncCall(_,params,_):
   (@!1!3,top,eval_params(params,@!1!3,1U))


mirCall(_,params):
   (@!1!3,top,eval_params(params,@!1!3,1U))


mirTailCall(_,_):
   ( [->top],top,[->top])


mirBeginProcedure(pars,locals):
   (init_locals(fill_pars(@!1!3,pars,@!3!3,1U),locals),top,[->top])


mirReturn(va,_):
   (@!1!3,value(va,@!1!3),[->top])


mirEndFuncCall(mirObjectAddr(_,res)):
   ( if is_volatile(res) then @!1!3 else @!1!3\[id(res) -> @!2!3] endif
     , @!2!3, [->top])

mirCallRet(pars,locals) :
   (built([->bot],@!1!3,pars,locals),@!2!3,[->top])

mirEndProcedure(pars,locals):
   (erase_var(erase_var(@!1!3,pars),locals),@!2!3,[->top])


default:
   (@!1!3,top,[->top])


SUPPORT

upd(f,flow,[!]) = f;
upd(f,flow,p::ps) =
      let f = upd(f,flow,ps); in
      case p of
            mirLocal(Id,_,_,_) => f\[myid(Id)->flow{myid(Id)}];
            mirParameter(Id,_,_,_,_) => f\[myid(Id)->flow{myid(Id)}];
            _ => f;
      endcase;
```

```
built(f,flow,pars,locals) = let a1 = upd(f,flow,pars);
                     in upd(a1,flow,locals);

comb(a,b) = a lub b;

erase_var(flow,[!]) = flow;
erase_var(flow,v::vs) =
   let f = erase_var(flow,vs); in
   case v of
      mirLocal(Id,_,_,_) => f\[myid(Id)->bot];
      mirParameter(Id,_,_,_,_) => f\[myid(Id)->bot];
      _ => f;
   endcase;


get_objinfo:: mirObject -> numbool;
get_objinfo(obj) =
   case obj of
      mirDataGlobal(id,_,v,_,_) => (myid(id),val-BOOL(v));
      mirLocal(id,_,v,_) => (myid(id),val-BOOL(v));
      mirParameter(id,_,v,_,_) => (myid(id),val-BOOL(v));
      mirRegister(id,_,v,_) => (myid(id),val-BOOL(v));
      _ => (0U,true);
   endcase;

is_volatile(obj) = get_objinfo(obj)!2!2;

id(obj) = get_objinfo(obj)!1!2;

eval_params:: *mirEXPR, const_fkt ,unum -> const_fkt;
eval_params([!],_,_) = [->top];
eval_params(e::es,flow,n) = eval_params(es,flow,n+1U)\[n->value(e,flow)];


init_locals::const_fkt,*mirObject -> const_fkt;
init_locals(f,[!]) = f;
init_locals(f,o::os) =
   let f = init_locals(f,os); in
   case o of
      mirLocal(id,typ,vol,_) =>
         if val-BOOL(vol) then error("error in ccmir: volatile local found!\n")
         else
         if is_simple_type(typ) then f\[myid(id)->uninit()] else f endif
         endif;
      _ => error("error in ccmir: Local Variable not a mirLocal!\n");
   endcase ;


fill_pars:: const_fkt,*mirObject,const_fkt,unum -> const_fkt;
fill_pars(f,[!],_,_) = f;
fill_pars(f,o::os,par,n) =
   let f = fill_pars(f,os,par,n+1U); in
   case o of
      mirParameter(id,typ,vol,_,pa) =>
         if val-BOOL(vol) then  error("error in ccmir: Volatile Parameter!\n")
         else
         case pa of
            ByValue =>
               if is_simple_type(typ) then f\[myid(id)->par{n}] else f endif;
            _ => error("unexpected Parameter kind");
         endcase
         endif;
```

```
      _ => error("unexpected Parameter Type");
   endcase;


is_simple_type:: mirTYPE -> bool;
is_simple_type(t) =
   case t of
      mirBoolean() => true;
      mirInteger(_) => true;
      mirReal() => true;
      _ => false;
   endcase;


type:: mirEXPR -> mirTYPE;
type(x) =
   case x of
      mirNoExpr(t) => t;
      mirIntConst(t,_) => t;
      mirRealConst(t,_) => t;
      mirBoolConst(t,_) => t;
      mirAddrConst(t,_) => t;
      mirArrayCom(t,_) => t;
      mirRecordCom(t,_) => t;
      mirUnionCom(t,_,_) => t;
      mirTypeSize(t,_) => t;
      mirObjectAddr(t,_) => t;
      mirContent(t,_) => t;
      mirSubscript(t,_,_) => t;
      mirMember(t,_,_) => t;
      mirBitExtract(t,_,_,_) => t;
      mirBitInsert(t,_,_,_,_) => t;
      mirBoolVal(t,v1,_,_) => t;
      mirConvert(t,v1,_) => t;
      mirCast(t,v1) => t;
      mirCheckReal(t,v1) => t;
      mirCheckRange(t,v1) => t;
      mirAbs(t,v1) => t;
      mirNeg(t,v1) => t;
      mirNot(t,v1) => t;
      mirAddrPlus(t,v1,v2,_) => t;
      mirAddrDiff(t,v1,v2,_) => t;
      mirCompare(t,v1,v2,_,_) => t;
      mirPlus(t,v1,v2,_) => t;
      mirDiff(t,v1,v2,_) => t;
      mirMult(t,v1,v2,_) => t;
      mirShiftLeft(t,v1,v2,_) => t;
      mirShiftRight(t,v1,v2,_) => t;
      mirShiftRightSign(t,v1,v2,_) => t;
      mirAnd(t,v1,v2,_) => t;
      mirOr(t,v1,v2,_) => t;
      mirXor(t,v1,v2,_) => t;
      mirDiv(t,v1,v2,_,_) => t;
      mirQuo(t,v1,v2,_,_) => t;
      mirMod(t,v1,v2,_,_) => t;
      mirRem(t,v1,v2,_,_) => t;
   endcase;


// uninitialized value
uninit:: -> va;
uninit() = lift(up-const_val-4(nix));
```

```
uninitfunc:: -> const_fkt;
uninitfunc() = [->uninit()];

is_signed(mirInteger(s)) = val-BOOL(s);
is_signed(_) = error("is_signed");

//  interpreter for expressions
value :: mirEXPR,const_fkt -> va;
value(e,f) =
   if !is_simple_type(type(e)) then uninit() else
   case e of
      mirNoExpr(t) => top;
      mirIntConst(t,x) => lift(
         if is_signed(t) then up-const_val-2(val-UNIV_INT(x))
            else up-const_val-5(unum(val-UNIV_INT(x))) endif);
      mirRealConst(t,x) => lift(up-const_val-3(val-UNIV_REAL(x)));
      mirBoolConst(t,x) => lift(up-const_val-1(val-BOOL(x)));
      mirAddrConst(t,x) => uninit();
      mirArrayCom(t,x) => uninit();
      mirUnionCom(t,x,y) => uninit();
      mirTypeSize(t,x) => top;
      mirObjectAddr(t,x) => uninit();
      mirContent(t,mirObjectAddr(t1,obj)) => f{id(obj)};
      mirContent(t,x) => top;
      mirSubscript(t,x,y) => top;
      mirMember(t,x,y) => uninit();
      mirBitExtract(_,_,_,_) => top;
      mirBitInsert(_,_,_,_,_) => top;
      mirBoolVal(_,va,tr,fa) =>
         let d <= value(va,f); in
            if is-const_val-1(d) then
            lift(up-const_val-2(val-UNIV_INT(
            if down-const_val-1(d)
               then tr
               else fa
            endif
            )))
            else if is-const_val-4(d) then lift(d) else
               error("unknown type in mirBoolVal")
            endif endif;
      mirConvert(t,va,round) =>
         if ! is_simple_type(type(va))
            then top
            else
               let d <= value(va,f); in
               if is-const_val-2(d) then
                  case t of
                     mirReal() =>
                        lift(up-const_val-3(real(down-const_val-2(d))));
                     mirInteger(signed) =>
                        if val-BOOL(signed) then
                           lift(d)
                        else
                           lift(up-const_val-5(unum(abs(down-const_val-2(d)))))
                        endif;
                     _ => error("Unknown Type convert");
                  endcase
               else if is-const_val-5(d) then
                  case t of
                     mirReal() =>
                        lift(up-const_val-3(real(down-const_val-5(d))));
```

```
                mirInteger(signed) =>
                   if val-BOOL(signed) then
                      lift(up-const_val-2(snum(down-const_val-5(d))))
                   else
                      lift(d)
                   endif;
                _ => error("Unknown Type convert");
             endcase
          else if is-const_val-4(d) then lift(d)
          else if is-const_val-3(d) then
             let x = down-const_val-3(d); in
             let s = sgn(x);
            w = sgn(x) >= 0;
            r = abs(x); in
             let p = unum(r+.5);
            c = unum(r); in
              let l = if real(c)=r then c else c+1U endif;
             in
             let erg =
             case round of
                Truncation() => c;
                Nearest() => p;
                Floor() => if w then c else l endif ;
                Ceiling() => if w then l  else c endif;
             endcase; in
                case t of
                   mirReal() => lift(d);
                   mirInteger(signed) =>
                      if val-BOOL(signed)
                      then lift(up-const_val-2(s*snum(erg)))
                      else lift(up-const_val-5(erg))
                      endif;
                endcase
             else error("Unknown type-convert")
             endif endif endif endif endif ;
mirCast(_,va) => top;
mirAbs(_,exp) =>
   let erg <= value(exp,f); in
   if is-const_val-2(erg) then lift(up-const_val-2(abs(down-const_val-2(erg))))
   else if is-const_val-3(erg) then lift(up-const_val-3(abs(down-const_val-3(erg))))
   else if is-const_val-5(erg) then lift(up-const_val-5(abs(down-const_val-5(erg))))
   else if is-const_val-4(erg) then lift(erg)
   else error("Wrong type") endif endif endif endif;
mirNeg(_,va) =>
   let v <= value(va,f); in
   if is-const_val-2(v) then lift(up-const_val-2(-down-const_val-2(v)))
   else if is-const_val-3(v) then lift(up-const_val-3(-down-const_val-3(v)))
   else if is-const_val-5(v) then top // minus unsigned !
   else if is-const_val-4(v) then lift(v)
   else error("Wrong type in neg") endif endif endif endif;
mirNot(_,va) =>
   let v <= value(va,f); in
   if is-const_val-1(v) then lift(up-const_val-1(!down-const_val-1(v)))
   else error("Wrong type in not") endif;
mirCompare(_,v1,v2,_,rel) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then
         let
```

```
            i1 = down-const_val-2(v1);
            i2 = down-const_val-2(v2);
         in up-const_val-1(
            case rel of
               Less() => i1 < i2;
               LessEqual() => i1 <= i2;
               Greater() => i1 > i2;
               GreaterEqual() => i1 >= i2;
               Equal() => i1 = i2;
               UnEqual() => i1 != i2;
            endcase
         )
         else
            if is-const_val-3(v1) && is-const_val-3(v2)
            then
            let
               i1 = down-const_val-3(v1);
               i2 = down-const_val-3(v2);
            in up-const_val-1(
               case rel of
                  Less() => i1 < i2;
                  LessEqual() => i1 <= i2;
                  Greater() => i1 > i2;
                  GreaterEqual() => i1 >= i2;
                  Equal() => i1 = i2;
                  UnEqual() => i1 != i2;
               endcase
            )
            else
               if is-const_val-5(v1) && is-const_val-5(v2)
               then
               let
                  i1 = down-const_val-5(v1);
                  i2 = down-const_val-5(v2);
               in up-const_val-1(
                  case rel of
                     Less() => i1 < i2;
                     LessEqual() => i1 <= i2;
                     Greater() => i1 > i2;
                     GreaterEqual() => i1 >= i2;
                     Equal() => i1 = i2;
                     UnEqual() => i1 != i2;
                  endcase
               )
               else
                  if is-const_val-4(v1) || is-const_val-4(v2)
                     then v1
                     else error("wrong type in Compare")
                  endif
               endif
            endif
         endif
   );
mirPlus(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(down-const_val-2(v1)+down-const_val-2(v2))
         else
         if is-const_val-5(v1) && is-const_val-5(v2)
```

```
            then up-const_val-5(down-const_val-5(v1)+down-const_val-5(v2))
            else
               if is-const_val-3(v1) && is-const_val-3(v2)
                  then up-const_val-3(down-const_val-3(v1)+down-const_val-3(v2))
                  else
                     if is-const_val-4(v1) || is-const_val-4(v2)
                     then up-const_val-4(nix)
                     else error("wrong types")
                  endif
               endif
         endif
      endif
   );
mirDiff(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(down-const_val-2(v1)-down-const_val-2(v2))
         else
            if is-const_val-3(v1) && is-const_val-3(v2)
               then up-const_val-3(down-const_val-3(v1)-down-const_val-3(v2))
               else
               if is-const_val-5(v1) && is-const_val-5(v2)
                  then up-const_val-5(down-const_val-5(v1)-down-const_val-5(v2))
                  else
                     if is-const_val-4(v1) || is-const_val-4(v2)
                     then up-const_val-4(nix)
                     else error("wrong types")
                  endif
               endif
            endif
      endif
   );
mirMult(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(down-const_val-2(v1)*down-const_val-2(v2))
         else
            if is-const_val-3(v1) && is-const_val-3(v2)
               then up-const_val-3(down-const_val-3(v1)*down-const_val-3(v2))
               else
            if is-const_val-5(v1) && is-const_val-5(v2)
               then up-const_val-5(down-const_val-5(v1)*down-const_val-5(v2))
               else
                  if is-const_val-4(v1) || is-const_val-4(v2)
                     then up-const_val-4(nix)
                  else error("wrong types")
               endif
            endif
            endif
      endif
   );
mirAnd(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in
```

```
      if is-const_val-1(v1) && is-const_val-1(v2)
         then lift(up-const_val-1(down-const_val-1(v1) && down-const_val-1(v2)))
         else
         if is-const_val-2(v1) && is-const_val-2(v2)
            then top
            else
         if is-const_val-5(v1) && is-const_val-5(v2)
            then top
            else
            if is-const_val-4(v1) || is-const_val-4(v2)
               then lift(up-const_val-4(nix))
               else error("wrong types")
            endif
         endif
         endif
      endif ;
mirOr(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in
      if is-const_val-1(v1) && is-const_val-1(v2)
         then lift(up-const_val-1(down-const_val-1(v1) || down-const_val-1(v2)))
         else
         if is-const_val-2(v1) && is-const_val-2(v2)
            then top
            else
         if is-const_val-5(v1) && is-const_val-5(v2)
            then top
            else
            if is-const_val-4(v1) || is-const_val-4(v2)
               then lift(up-const_val-4(nix))
               else error("wrong types")
            endif
         endif
      endif
   endif;
mirXor(_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in
      if is-const_val-1(v1) && is-const_val-1(v2)
         then lift(up-const_val-1(
            let
               b1 = down-const_val-1(v1);
               b2 = down-const_val-1(v2);
            in (b1 || b2) && !(b1 && b2)))
         else
         if is-const_val-2(v1) && is-const_val-2(v2)
            then top
            else
         if is-const_val-5(v1) && is-const_val-5(v2)
            then top
            else
            if is-const_val-4(v1) || is-const_val-4(v2)
               then lift(up-const_val-4(nix))
               else error("wrong types")
            endif
         endif
         endif
      endif;
```

```
mirDiv(_,_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(
            let
               e = real(down-const_val-2(v1))/real(down-const_val-2(v2));
            in
            trunc(if e<0.0 then e-1.0 else e endif)
            )
         else
      if is-const_val-5(v1) && is-const_val-5(v2)
         then up-const_val-5(
            let
               e = real(down-const_val-5(v1))/real(down-const_val-5(v2));
            in
               unum(trunc(if e<0.0 then e-1.0 else e endif))
            )
         else
         if is-const_val-3(v1) && is-const_val-3(v2)
            then up-const_val-3(down-const_val-3(v1)/down-const_val-3(v2))
            else
            if is-const_val-4(v1) || is-const_val-4(v2)
               then up-const_val-4(nix)
               else error("wrong types")
            endif
         endif
      endif
      endif
   );
mirQuo(_,_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(down-const_val-2(v1)/down-const_val-2(v2))
         else
      if is-const_val-5(v1) && is-const_val-5(v2)
         then up-const_val-5(down-const_val-5(v1)/down-const_val-5(v2))
         else
         if is-const_val-3(v1) && is-const_val-3(v2)
            then up-const_val-3(down-const_val-3(v1)/down-const_val-3(v2))
            else
            if is-const_val-4(v1) || is-const_val-4(v2)
               then up-const_val-4(nix)
               else error("wrong types")
            endif
         endif
      endif
      endif
   );
mirMod(_,_,v1,v2,_) =>
   let
      v1 <= value(v1,f);
      v2 <= value(v2,f);
   in lift(
      if is-const_val-2(v1) && is-const_val-2(v2)
         then up-const_val-2(sgn(down-const_val-2(v1))*(abs(down-const_val-2(v1))
            %abs(down-const_val-2(v2))))
```

```
              else
            if is-const_val-5(v1) && is-const_val-5(v2)
              then up-const_val-5(down-const_val-5(v1)%down-const_val-5(v2))
              else
              if is-const_val-3(v1) && is-const_val-3(v2)
                then up-const_val-3(down-const_val-3(v1)%down-const_val-3(v2))
                else
                if is-const_val-4(v1) || is-const_val-4(v2)
                  then up-const_val-4(nix)
                  else error("wrong types")
                endif
              endif
              endif
            endif);
      mirShiftLeft(_,_,_,_) => top;
      mirShiftRight(_,_,_,_) => top;
      mirShiftRightSign(_,_,_,_) => top;
      mirRem(_,_,v1,v2,_) =>
        let
          v1 <= value(v1,f);
          v2 <= value(v2,f);
        in lift(
          if is-const_val-2(v1) && is-const_val-2(v2)
            then up-const_val-2(sgn(down-const_val-2(v2))*(abs(down-const_val-2(v1))
              %abs(down-const_val-2(v2))))
            else
          if is-const_val-5(v1) && is-const_val-5(v2)
            then up-const_val-5(down-const_val-5(v1)%down-const_val-5(v2))
            else
            if is-const_val-3(v1) && is-const_val-3(v2)
              then up-const_val-3(down-const_val-3(v1)%down-const_val-3(v2))
              else
              if is-const_val-4(v1) || is-const_val-4(v2)
                then up-const_val-4(nix)
                else error("wrong types")
              endif
            endif
          endif
          endif
        );
   _ => top;
endcase
endif;

myid::INT -> unum;
myid(id) =
   let x = val-INT(id); in
   if (x<0) then error("Negative id found!")
   else unum(x)
   endif;
```

Anhang B

Meßergebnisse

Für diese Messungen wurden zwei Analysatoren mit PAG generiert: eine vollständige Konstantenpropagation und eine Copy Konstantenpropagation (vgl. Kapitel 9 auf Seite 91). Diese wurden in Verbindung mit dem **COMPARE** ANSI C Compiler auf verschiedene Programme angewandt. Die nachfolgende Tabelle enthält für jedes Testprogramm zwei Zeilen. In der ersten Zeile werden die Daten für die volle und in der zweiten Zeile die für die Copy Konstantenpropagation aufgeführt. Daten, die beiden gemeinsam sind wie z.B. die Anzahl der Prozeduren, werden nur einmal aufgeführt. Die Bedeutung der Spalten ist im einzelnen:

Programm:	Name des analysierten Programms. Es handelt sich ausschließlich um Standardsoftware, die in jedem gut sortierten `ftp` Archiv zu finden ist
proc:	Anzahl der Prozeduren im Programm
Knoten:	Die Anzahl der Knoten des Kontrollflußgraphen, der vom Compiler (nicht von PAG) erzeugt wurde. Diese Zahl hängt vom verwendeten Kontrollflußanalysator ab
obj:	Anzahl der Objekte, die untersucht wurden. Auch diese Zahl kann durch das Frontend beeinflußt werden, falls dort temporäre Objekte erzeugt wurden
IPC:	Die Zahl der interprozeduralen Konstanten. Das sind konstante Objekte an Prozedurköpfen $\sum_{procedure\ p} \lvert\{obj \mid (\bigsqcup_{i=1,\ldots,Arity(p)} flow(p)[i])(obj) \neq \bot, \top\}\rvert$ Diese Zahl ist besondesr geeignet, um die Qualität der interprozeduralen Analyse abzuschätzen
CON:	Die Summe aller konstanten Objekte an allen Knoten $\sum_{flownode\ p} \lvert\{obj \mid (\bigsqcup_{i=1,\ldots,Arity(p)} flow(p)[i])(obj) \neq \bot, \top\}\rvert$

FOL: Die Anzahl der Faltungen, die in einem nachgeschalteten Konstantenfalter durch das Ersetzen von konstanten Objekten entstehen (auch diese Zahl ist vom Frontend abhängig)

$\#f_{obj}$: x/y, x ist die Zahl der Faltungen von konstanten Objekten, die Variablen des Quellprogramms repräsentieren (Frontend unabhängig) und y ist die Zahl der vom Frontend eingeführten Objekte

$\#_{rest}$: Faltungen innerhalb von Array Zugriffen

$\#_{eva}$: Faltungen in vom Frontend eingeführten Codes

Schritte: Die Zahl der Anwendungen von Transferfunktionen auf ein einzelnes Datenflußelement

Programm	proc	Knoten	obj	IPC	CON	FOL	$\#f_{obj}$	$\#_{rest}$	$\#_{eva}$	Schritte
`cdecl`	32	3209	835	7	1010	106	2/80	354	3	55160
				7	1010	106	3/80	354	3	55187
`ed`	46	3448	784	7	1481	54	7/59	132	6	44628
				9	1509	62	11/59	132	6	44333
`fft`	7	638	195	3	439	26	0/19	52	10	17772
				3	439	26	0/19	52	10	17896
`find`	210	13191	2704							
				17	6358	1196	29/145	330	99	701606
`flex`	128	11974	2301	49	4926	404	59/335	1008	64	140768
				49	4980	404	59/336	1008	64	138960
`flops`	2	640	139	9	5820	217	90/47	202	38	28971
				9	6209	294	161/61	955	150	30561
`gzip`	99	8332	1740	11	4353	594	36/250	995	153	137655
				11	4358	600	37/180	995	153	145863
`heap`	3	361	94	8	1432	24	13/34	42	13	6777
				8	1432	24	13/34	42	13	7765
`linpack`	12	1196	278	6	587	287	28/51	392	98	15430
				6	610	295	32/55	392	98	15628
`ratfor`	51	3200	762	0	934	152	6/54	287	36	101901
				0	934	152	6/54	287	36	101952
`sieve`	1	58	15	0	21	3	0/4	6	2	150
				0	21	3	0/4	6	2	157
`twig`	81	4525	1085	6	1332	109	3/64	342	33	35437
				6	1332	109	3/64	342	33	36122
`xmodem`	30	4460	1157	38	4849	181	6/164	492	28	98095
				38	4963	191	12/166	492	34	98719
`acker`	2	35	14	0	0	0	0/0	1	0	201
				0	0	0	0/0	1	0	220
`base`	2	289	100	2	332	52	8/5	88	12	3276
				2	334	52	8/5	88	12	3368
`clinpack`	12	1257	285	6	668	287	28/51	392	98	16033
				6	691	295	32/55	392	98	16231
`cloops`	126	3212	620	127	13341	956	56/177	1185	192	301094
				131	14436	960	60/179	1185	203	301041

Programm	proc	Knoten	obj	IPC	CON	FOL	$\#f_{obj}$	$\#_{rest}$	$\#_{eva}$	Schritte
`dhry`	13	662	239	8	279	9	2/13	97	2	17262
				8	286	10	3/14	97	2	17282
`search`	3	89	34	2	189	6	7/8	9	0	272
				2	189	6	7/8	9	0	283
`whet`	7	492	121	3	397	75	16/44	81	23	36078
				3	448	132	40/49	81	33	36961

Literaturverzeichnis

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy Compiler Phase Embedding with the Cosy Compiler Model. In *6th International Conference for Compiler Construction in Edinburgh*, volume LNCS786. Springer, 1994.

[All70] Frances E. Allen. Control flow analysis. In Proceedings of a Symposium on Compiler Optimization, volume 5, pages 1–19, July 1970.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. *International Static Analysis Symposium, SAS '95*, September 1995. To appear in LNCS Springer-Verlag.

[Aßm93] Uwe Aßmann. *Optimix – A generator for dataflow equitation systems*, April 1993.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[Bar77] J.M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 119–131, Los Angeles, CA, January 1977.

[Bar78] J.M. Barth. A practical interprocedural data flow analysis algorithm. *Journal of the ACM*, 21(9):724–736, September 1978.

[Bar83] G. Barth. Interprocedural data flow systems. In *6th GI Conference on Theoretical Computer Science, Dortmund, Germany*, number 145 in LNCS, pages 49–59. Springer-Verlag, 1983.

[Car85] Luca Cardelli. Basic polymorphic typechecking. *Polymorphism Newsletter*, 2(1), January 1985. Also appeared as Computing Science Technical Report 119, AT&T Bell Laboratories, Murray Hill, NJ.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[CC92a] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, NM, January 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, volume 21, pages 152–161, Palo Alto, CA, June 1986.

[Coc70] John Cocke. Global common subexpression elimination. In Proceedings of a Symposium on Compiler Optimization, volume 5, pages 20–24, July 1970.

[COM90] The COMPARE Consortium. Clax – A Compare Language Example. Rel. 1.2, 1990.

[COM93a] The COMPARE Consortium. CCMIR Definition of a Program Analyzer Generator. Ref. ACE-1038-ccmir2, rel. 6.1, 1993.

[COM93b] The COMPARE Consortium. Design of a Program Analyzer Generator. Ref. USAAR-1017-ana, rel. 1.0, 1993.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedual data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[DRZ92] D.M. Dhamdhere, B.K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 212–223, San Francisco, CA, June 1992.

[GT93] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, 1993.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.

[JM75] Neil D. Jones and Steven S. Muchnick. Even simple programs are hard to analyze. In *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages*, pages 106–118, January 1975.

[Jon87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, London, UK, 1987.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[Kno93] Jens Knoop. *Optimal interprocedural program optimization: A new framework and its application.* PhD thesis, Christian-Albrechts-Universität zu Kiel, 1993.

[KR90] Brian W. Kernigham and Dennis M. Ritchie. *Programmieren in C.* Carl Hanser, München, 2 edition, 1990.

[KRS92] Jens Knoop, Oliver Ruething, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[KRS95] Jens Knoop, Oliver Rütting, and Bernhard Steffen. A toolkit for constructing optimal interprocedural data flow analyses. 1995.

[KS92] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, pages 125–140. Springer-Verlag LNCS 641, 1992. also appeard as Technical Report of University of Aachen no. 91-27.

[KU77] J.B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Mar95] Florian Martin. *PAG Reference Manual.* Universität des Saarlandes, 1995.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* Springer Verlag, 1984. ISBN 3-540-13641-X.

[MFS79] Ronald J. Mintz, Gerald A. Fischer, Jr., and Micha Sharir. The design of a global optimizer. In *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction*, volume 14, pages 226–234, August 1979.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[NN92] H.R. Nielson and F. Nielson. Bounded fixed point iteration. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 71–82, Albuquerque, NM, January 1992.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.

[Ros93] Mads Rosendahl. Higher-order chaotic iteration sequences. In *PLILP'93, Tallinn, Estonia*, number 714 in LNCS, pages 332–345. Springer-Verlag, 1993.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[SRH95] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with application to constant propagation. In *TAPSOFT'95, Arhus, Denmark*, LNCS. Springer-Verlag, 1995.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A tool for building optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, San Francisco, CA USA, [7] 1992. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 27, number 7.

[VF92] G.V. Venkatesch and Charles N. Fischer. Spare: A development environment For Program Analysis Algorithms. In *IEEE Transactions on Software Engineering*, volume 18, 1992.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer Verlag, April 1992. ISBN 3-54055704-0.

[WZ85] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, LA, January 1985.

[YH93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, Charleston, South Carolina, January 1993.

[ZC91] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM Press, New York, NY, 1991.