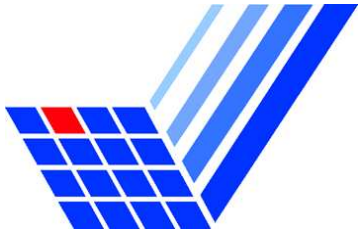
Diplomarbeit

Entwicklung einer plattformunabhängigen, kontext-sensitiven Aliasanalyse für das ICD-C Compiler-Framework

Holger Bihr



Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

10. November 2005

Betreuer:

Dipl.-Inf. Robert Pyka
Prof. Dr. Peter Marwedel

Holger Bihr
Kopernikusstraße 4a
46147 Oberhausen

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die zunehmenden Ansprüche an Computersysteme führen zu neuen Herausforderungen für die Entwicklung dieser Systeme. In Kombination mit sich schnell ändernden Marktbedürfnissen hat dies dazu geführt, dass ein immer größerer Teil der Funktionalität in Software implementiert wird. Das liegt darin begründet, dass eine Änderung der Software kürzere Entwicklungszyklen erlaubt als eine Änderung der Hardware und dass auch bereits bestehende Geräte durch ein Softwareupdate auf den aktuellen Stand gebracht werden können.

Um die erwähnten kurzen Entwicklungszyklen zu erreichen ist der Einsatz von Hochsprachen für die Programmierung zwingend erforderlich. Ein wesentlicher Vorteil dieser Sprachen besteht darin, dass die Quelltexte mit geringem Aufwand auf verschiedene Hardwarearchitekturen portiert werden können; üblicherweise genügt es, den Programmquelltext mit einem *Compiler* für die neue Prozessorarchitektur zu übersetzen. Eine solche Portierung auf der Ebene von Assemblercode dagegen ist praktisch nicht möglich. Diese Portabilität kann nun dazu genutzt werden, um Softwarekomponenten wiederzuverwenden. Diese Vorgehensweise verkürzt direkt die Entwicklungszyklen.

Der Anspruch an Compiler ist das Erzeugen von möglichst effizientem Programmcode [Muc97]. Darunter versteht man, dass die erzeugten Maschinenprogramme möglichst wenig Speicherplatz benötigen und die Ausführungszeit minimiert wird. Insbesondere im Bereich der eingebetteten Systeme ist diese Effizienz von sehr großer Bedeutung [Mar03]. So wird beispielsweise für viele Anwendungen Echtzeitfähigkeit verlangt. Dadurch kann es passieren, dass ein Programm, das stark optimiert wurde, alle Zeitschranken einhält, wenn es auf einem bestimmten Prozessor ausgeführt wird, in schwach optimierter Form jedoch einen leistungsfähigeren Prozessor - etwa mit einer höheren Taktrate und ansonsten identischer Architektur - benötigt. Analoges gilt für den Speicherverbrauch, hier kann ein schwach

optimiertes Programm einen unnötig großen Hauptspeicher erfordern. Aus diesen Beobachtungen lässt sich ableiten, dass schwach optimierte Programme zwei unerwünschte Faktoren - Kosten sowie den insbesondere bei mobilen Geräten wichtigen Energieverbrauch - vergrößern und somit die Marktakzeptanz dieses Gerätes verringern. Daher wurden vielfältige Algorithmen entwickelt, die helfen sollen, Programmcode in eine möglichst optimale Form zu überführen. Dazu gehört auch die *Aliasanalyse*. Ziel dieser Analyse ist es, herauszufinden, auf welche Speicherbereiche *Zeigervariablen* zeigen können. Diese Information ist von grundlegender Bedeutung, denn viele andere Algorithmen können durch diese Information den Programmcode aggressiver optimieren.
Bisherige Algorithmen zur Aliasanalyse arbeiten jedoch häufig relativ ungenau, da sie nicht alle Informationen aus dem Programmquelltext verwenden. Das führt dazu, dass Aliasse angenommen werden, die es im realen Programm nicht gibt. Dadurch werden die Optimierungsmöglichkeiten für andere Algorithmen zur Programmoptimierung unnötig eingeschränkt. Beispiele für nicht genutzte Informationen sind etwa der Kontrollfluss sowie komplexe Datenstrukturen; in letzteren werden oft nicht die einzelnen Felder unterschieden.
Dies führte zu der Motivation, durch möglichst intensive Nutzung der Informationen aus dem Programmquelltext die Genauigkeit der Ergebnisse gegenüber bisherigen Techniken zu verbessern.

1.2 Zielsetzung

Die Zielsetzung dieser Arbeit besteht in der Entwicklung einer Aliasanalyse für *C* Programme. Die Ergebnisse dieser Analyse sollen durch ein definiertes Interface von einem Compiler aus abgefragt werden können. Ebenso sollte die Leistungsfähigkeit der implementierten Aliasanalyse untersucht werden.

Im einzelnen ergeben sich daraus folgende Ziele für diese Diplomarbeit:

1. Das Hauptziel ist die Entwicklung der Aliasanalyse. Diese Analyse soll Programme, die in der Hochsprache *C* geschrieben sind, analysieren können. Die Sprache *C* wurde gewählt, weil sie insbesondere bei eingebetteten Systemen sehr weit verbreitet ist. Die Verwendbarkeit für Hochsprachen impliziert dabei, dass während der Analyse keine architekturabhängigen Annahmen gemacht werden dürfen. Weiterhin sollte der Analysealgorithmus möglichst genaue Ergebnisse liefern, was nur durch eine globale Programmanalyse möglich ist.

2. Bewertung der Qualität der Analyse
   Die Qualität der Analyse ergibt sich durch die erkannten Aliasse. Dabei ist zu beachten, dass die exakte Erkennung von Aliasbeziehungen im Allgemeinen nicht möglich ist [Hin01]. Mit konservativen Annahmen lässt sich jedoch

eine Menge von Aliasbeziehungen errechnen, die alle tatsächlichen Aliasse enthält. Da die Größe dieser Menge jedoch nicht aussagekräftig für die Qualität der Analyse ist, wird der Qualitätsgewinn anderer Optimierungsalgorithmen gemessen, der sich einstellt, wenn die Ergebnisse der Aliasanalyse genutzt werden. Diese Messungen sollen, um die praktische Relevanz aufzuzeigen, mit realen Programmen durchgeführt werden.

1.3 Aufbau der Arbeit

Im Kapitel 2 werden zunächst die Grundlagen der Aliasanalyse erläutert. Dazu gehört eine Beschreibung über die Möglichkeiten zur Nutzung von Zeigervariablen in *C* und die Vorstellung einiger bereits bestehender Analysetechniken. Diese Techniken werden bewertet und miteinander verglichen.

Eine genaue Beschreibung des implementierten Algorithmus findet sich in Kapitel 3. Die Entwurfsentscheidungen und das Interface der Aliasanalyse werden ebenfalls in diesem Kapitel aufgeführt.

Die Auswahl der Benchmarks und die Optimierungsalgorithmen, die von der Aliasanalyse profitieren sollen, werden im Kapitel 4 beschrieben. Diese Beschreibung enthält jeweils eine Erläuterung des Optimierungsziels, eine Darstellung des jeweiligen Algorithmus sowie die ermittelten Benchmarkergebnisse. Lediglich für die nur skizzierten Anwendungen, die im Kapitel 4.5 erwähnt sind, wird neben einer algorithmischen Beschreibung nur theoretisch aufgezeigt, wie die Ergebnisse der Aliasanalyse genutzt werden können.

Im Anhang sind die Benchmarkergebnisse noch einmal in konzentrierter Form aufgelistet. Ebenso findet sich dort eine kurze Dokumentation der in Kapitel 4 genannten Anwendungen.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die Grundlagen der Aliasanalyse. Im Unterkapitel 2.1 wird ein kurzer Überblick über die Möglichkeit gegeben, Aliasse in der Programmiersprache *C* zu erzeugen. Das Unterkapitel 2.2 enthält eine Übersicht und Bewertung verschiedener Analysetechniken. Es folgt eine kurze Beschreibung, welche Form der Aliasanalyse in heute verwendeten Compilern realisiert ist. Abschließend wird dargelegt, welche Zugriffsmöglichkeiten es im Compiler für die Aliasbeziehungen gibt.

2.1 Zeiger und Aliasing in C

Die Programmiersprache *C* bietet vielfältige Möglichkeiten zur Verwendung von Zeigern [KR88] [HS02]. Dabei versteht man unter Zeigern eine Variable, die die Speicheradresse einer Variablen enthält. Bei der Daklaration einer Zeigervariablen wird jeweils ein Basistyp angegeben. Dieser Basistyp hat einen Einfluss auf das Verhalten der Zeigervariablen, schränkt die Menge der möglichen Zielvariablen jedoch nicht auf solche des Basistyps ein. Das folgende Beispiel zeigt eine Deklaration eines Zeigers mit dem Basistyp int. Das * deklariert die Variable als Zeiger:

```
int *zeiger;
```

C definiert unäre Operatoren zur Nutzung von Zeigern. Das sind der Adressoperator &, mit dem die Speicheradresse einer Variablen bestimmt werden kann, und der Dereferenzierungsoperator *, mit dem die Zielvariable eines Zeigers angesprochen werden kann. Das folgende Beispiel weist dem Zeiger p die Adresse der Variablen v1 zu. Anschließend wird p dereferenziert und so der Inhalt der Zielvariablen - in diesem Fall v1 - der Variable v2 zugewiesen:

```
int *p, v1, v2;
p=&v1;
v2=*p;
```

Listing 2.1: Zeigervariablen

Während der Adressoperator nur auf der rechten Seite einer Zuweisung verwendet werden kann, kann der Dereferenzierungsoperator auch auf der linken Seite verwendet werden. Zeiger lassen sich ebenfalls über eine Zuweisung von Konstanten initialisieren. Diese Vorgehensweise ist beispielsweise sinnvoll, wenn absolute Speicheradressierung genutzt werden soll. Dabei wird die Zuweisung des Wertes 0 dazu verwendet, um anzuzeigen, dass ein Zeiger keine gültige Zielvariable besitzt.

Vektoren

Vektoren sind mit Zeigern eng verwandt, eine Vektoroperation kann in eine Zeigeroperation umgewandelt werden. Eine Deklaration für einen Vektor definiert einen Speicherbereich der so groß ist, dass so viele Elemente des Basistyps wie spezifiziert in diesen Bereich passen. Das folgende Beispiel deklariert einen Vektor mit 10 Elementen vom Basistyp int.

```
int a[10];
```

Auf die einzelnen Elemente kann mit dem Index-Operator zugegriffen werden, dabei hat das erste Element den Index 0. Der Ausdruck a[4] beispielsweise greift auf das fünfte Element im Vektor a zu. Diese Ausdrücke können ebenfalls durch Zeiger ausgedrückt werden. So hat das Symbol a im obigen Beispiel den Typ int *, es ist also ein Zeiger. Dieser Zeiger zeigt immer auf das erste Element im Vektor, also a[0]. Mit den auf Zeigern definierten binären Operationen + und - ist es möglich, auf andere Elemente des Vektors zuzugreifen. Dabei wird in diesen Ausdrücken jedoch nicht die Adresse um den angegebenen Offset erhöht bzw. verringert, sondern um diesen Betrag, multipliziert mit der Größe des Basistyps. So ist der Ausdruck *(a+4) äquivalent zu a[4]. Dieser Zusammenhang gilt auch für multidimensionale Arrays. So deklariert int b[5][10] ein Symbol b vom Typ int **, das ist ein Zeiger auf einen weiteren Zeiger mit dem Basistyp int. Die Ausdrücke b[x][y] und *(*(b+x)+y) sind äquivalent, dabei stellen x und y Indexvariablen dar.
Der *C99*-Standard bietet Programmierern die Möglichkeit, die Vektorgröße nicht nur durch eine Konstante, sondern auch durch eine Variable zu beschreiben. Dies kann dazu führen, dass für den Compiler die Größe eines Vektors nicht erkennbar ist.
Da keine Prüfung der Gültigkeit der Arrayindizes vorgenommen wird, können bei multidimensionalen Vektoren mehrere Kombinationen von Indexwerten das gleiche Vektorelement adressieren, im obigen Beispiel etwa sind die Ausdrücke b[0][20] und b[1][10] äquivalent. Noch weitreichendere Möglichkeiten ergeben sich, wenn ein Vektor in eine struct eingebettet ist.

```
struct {
  int a[10];
  int b[10];
} s;
```

Listing 2.2: In eine struct eingebettete Vektoren

So ist es mit der im Listing 2.2 dargestellten Datenstruktur möglich, etwa mittels b in Kombination mit einem negativen Index auf ein Element von Vektor a zuzugreifen. Eine derartige Adressierung ist nur innerhalb von einer struct möglich, da die Anordnung von Variablen im Speicher erst durch den Linker festgelegt wird.

Funktionszeiger

C erlaubt indirekte Funktionsaufrufe durch Funktionszeiger. Dabei handelt es sich um Zeiger, die Funktionen im Programm als Zielwert haben. Da dem Compiler die deklarierten Funktionen bekannt sind, kann dieser unterscheiden, ob bei einem Vorkommen eines Funktionssymbols im Quelltext das Funktionssymbol oder dessen Adresse gemeint ist. Listing 2.3 zeigt verschiedene Ausdrücke mit identischem Verhalten:

```
void f() {
  /* ... */
}

void main() {
  void (*p)(void); /* Deklaration Funktionszeiger */

  p=f; /* p wird die Zielfunktion f zugewiesen */
  p=&f; /* p wird die Zielfunktion f zugewiesen */

  p(); /* Zielfunktion von p, hier f, wird aufgerufen */
  (*p)(); /* Zielfunktion von p, hier f, wird aufgerufen */
}
```

Listing 2.3: Funktionszeiger

Aliasse

Ein Alias ergibt sich, wenn zwei Ausdrücke die gleiche Speicherstelle bezeichnen. Das ist offensichtlich der Fall zwischen den beiden Ausdrücken v1 und *p aus Listing 2.1. Ebenso besteht ein Alias zwischen jeweils zwei äquivalenten Zugriffen auf das gleiche Vektorelement.
Eine weitere Möglichkeit, eine Aliasbeziehung zu erzeugen, ist die Verwendung einer union-Datenstruktur. Diese Struktur kann mehrere Elemente enthalten, die alle an der gleichen Adresse abgespeichert werden.

2.2 Analysetechniken

Die hier vorgestellten Techniken sollen einen Überblick über bisherige Ansätze zur Aliasanalyse geben. Diese Techniken unterscheiden sich teilweise deutlich in der Präzision der ermittelten Aliasbeziehungen, jedoch gehen im Allgemeinen genauere Ergebnisse mit einer schlechteren Skalierbarkeit einher.

2.2.1 Inclusion-/equality-based

Dieses Kriterium beschreibt die Komplexität der möglichen Aliasbeziehungen. Während bei einer *inclusion-based* Analyse Zuweisungen unidirektional betrachtet werden, wird in einer *equality-based* Analyse eine Zuweisung bidirektional ausgeführt. So lässt sich eine Gruppe von Zeigervariablen, die einander zugewiesen werden, als eine Klasse betrachten, denn die Menge der Zielvariablen ist für alle Zeiger dieser Klasse identisch. Weiterhin werden alle Zielvariablen einer Klasse von Zeigern zu einer einzigen Klasse zusammengefasst. Diese Vereinfachung hat natürlich den Nachteil, dass die Genauigkeit der Ergebnisse reduziert wird, wie sich am Beispiel von Listing 2.4 erkennen lässt. Abbildung 2.1 zeigt die ermittelten Aliasbeziehungen. Die Knoten von diesem Graph stellen die einzelnen Klassen von Zeigern dar, während die Aliasbeziehungen zwischen diesen Klassen als *Points-to*-Werte dargestellt sind; eine Kante von Klasse `A` zu Klasse `B` bedeutet, dass jede Zeigervariable aus Klasse `A` auf jede beliebige Variable aus Klasse `B` zeigen kann, d.h.,`B` ein *Points-to*-Wert von `A` ist. Der Kontrollfluss wird in diesem Beispiel nicht berücksichtigt. Wie sich im *Points-to*-Graphen erkennen lässt, nimmt dieses Analyseverfahren an, dass `a` auf `z` zeigen kann. Bei Betrachtung des zugehörigen Listings ist jedoch festzustellen, dass diese Beziehung nicht möglich ist.

```
a=&x;
b=&y;
if(p)
  y=&z;
else
  y=&x;
c=&y;
```

Listing 2.4: Beispiel für Equality-based Analyse

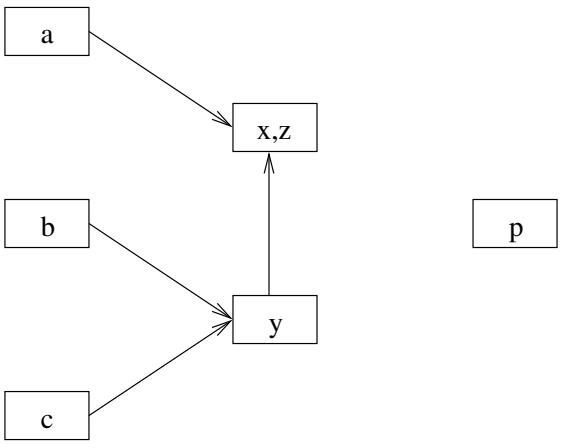


Abbildung 2.1: Aliasbeziehungen für Listing 2.4

Vorteil dieser Vorgehensweise ist jedoch, dass der Algorithmus nur eine geringe Rechenzeit benötigt. So beschränken sich die Operationen während der Analyse darauf,

- zwei Klassen von Zeigervariablen zusammenzufügen. Die Klassen der jeweiligen Zielvariablen müssen ebenfalls zusammengefügt werden.
- erstmalig eine Zeigervariable in eine Klasse aufzunehmen
- eine *Points-to*-Beziehung von einer Klasse zu einer anderen Klasse zu erzeugen.

Diese Operationen können auf *Union-find*-Datenstrukturen abgebildet werden. Dabei entsprechen die Klassen im Graphen den Mengen der *Union-Find*-Datenstruktur. Da aufgrund der bidirektionalen Betrachtung von Zuweisungen für diesen Algorithmus ein einziger Durchlauf durch den Programmcode genügt, ergibt sich, zusammen mit den - gemittelt - in annähernd konstanter Zeit ablaufenden *Union-Find*-Operationen [Weg], eine annähernd lineare Laufzeit für diesen Algorithmus [Ste96].

Bei einer *inclusion-based*-Analyse gibt es die oben genannten Vereinfachungen nicht. Zuweisungen werden unidirektional betrachtet und ein Zeiger kann auf beliebig viele andere Variablen zeigen. Eine Einteilung in Klassen wird nicht vorgenommen. Diese Form der Analyse geht auf Andersen [And94] zurück. Dabei werden nur diejenigen Anweisungen in einem Programm betrachtet, die einen Einfluss auf die Aliasbeziehungen haben können. Das sind im einzelnen:

- Zuweisung zu einer Zeigervariablen
- Benutzung des Adressoperators
- Benutzung des Dereferenzierungsoperators

Diese Operatoren können zwar in beliebig komplexen Anweisungen auftreten, jedoch ist es möglich, ein Programm so zu transformieren, dass die Anweisungen, die einen Einfluss auf die Aliasbeziehungen haben, in der folgenden Form vorliegen:

- `*x=e`
- `e=*x`
- `e=&x`
- `e=e2`

Dabei sind x, e, e2 Symbole. Auch dieser Algorithmus speichert Aliasbeziehungen in einem *Points-to*-Graphen. Der Algorithmus hat nun das Ziel, alle im Programm befindlichen - auch indirekten - Zuweisungen zu einer Zeigervariablen herauszufinden. Das lässt sich durch die folgenden Ableitungsregeln ausdrücken. Diese Regeln sind so dargestellt, dass wenn die oberhalb der Trennlinie aufgeführten Bedingungen erfüllt sind die unterhalb der Trennlinie genannte Eigenschaft gilt. Eine Beziehung $e1 \longrightarrow e2$ besagt, dass alle *Points-to*-Werte von e2 auch von e1 erreicht werden können. Das kann beispielsweise dadurch geschehen, dass $e2$ der Variablen $e1$ zugewiesen wird. P bezeichnet die Menge aller Anweisungen im Programm.

$\frac{x \longrightarrow \&y \wedge *x = e \in P}{y \longrightarrow e}$: Diese Regel besagt, dass wenn von x aus durch Dereferenzierung unter anderem auf die Variable y zugegriffen werden kann und das Programm eine Anweisung enthält, die der Zielvariable von x den Wert e zuweist, so muss angenommen werden, dass y der Wert von e zugewiesen wrid.

$\frac{x \longrightarrow \&y \wedge e = *x \in P}{e \longrightarrow y}$: Diese Regel deckt den Fall ab, dass wenn durch Dereferenzierung von x aus auf die Variable y zugegriffen werden kann und das Programm eine Anweisung enthält, die e die Zielvariable von x zuweist. In diesem Fall wird angenommen, dass e der Wert von y zugewiesen wird.

$\frac{e1 = e2 \in P}{e1 \longrightarrow e2}$: Diese Regel schreibt eine Zuweisung in eine Erreichbarkeitsbeziehung um.

$\frac{e1 \longrightarrow e2 \wedge e2 \longrightarrow e3}{e1 \longrightarrow e3}$: Durch diese Regel wird der transitive Abschluss der Erreichbarkeitsbeziehungen berechnet.

Die letzte Regel deckt indirekte Zuweisungen ab und bildet den transitiven Abschluss von Zuweisungen. In den Regeln werden nur bisher analysierte Zuweisungen betrachtet. Dies ist nicht ausreichend, wenn verzweigter Kontrollfluss vorhanden ist. Daher iteriert der Analysealgorithmus solange über alle Anweisungen aus P, bis die Menge der *Points-to*-Beziehungen einen Fixpunkt erreicht hat. Dabei wird die Menge der *Points-to*-Beziehungen so berechnet, dass eine Variable x genau dann auf eine Variable y zeigen kann, wenn eine der folgenden Bedingungen erfüllt ist:

- Das Programm enthält eine Anweisung der Form x=&y
- Das Programm enthält eine Variable z, eine Anweisung der Form z=&y und es gilt $x \longrightarrow z$.

Eine Movitation für die Verbesserung dieser Analysemethode war, dass die wiederholte Berechnung des transitiven Abschlusses sehr rechenintensiv ist. Als Verbesserung sei hier der Algorithmus von Heintze & Tardieu genannt [HT01]. In diesem Algorithmus werden die Anweisungen, die einen Einfluss auf die Aliasbeziehungen haben können, in drei Klassen unterteilt:

- Einfache Anweisungen der Form `x=y`
- Basiszuweisungen der Form `x=&y`
- Komplexe Zuweisungen der Form `*x=y` oder `y=*x`

Der Algorithmus arbeitet auf einem gerichteten Graphen, der für jede Variable `v` im Programm zwei Knoten besitzt, die mit n_v und n_{*v} bezeichnet werden. Zu jedem Knoten n_v wird eine Liste mit Basiselementen abgespeichert. Diese Liste enthält alle Variablen w, für die eine Zuweisung $v = \&w$ im Programm vorhanden ist. Weiterhin wird für jede einfache Anweisung `v=w` im Programm eine Kante von n_v nach n_w zum Graphen hinzugefügt. Dazu ist nur ein einfacher Durchlauf von P erforderlich, woraus sich eine lineare Laufzeit ergibt. Der iterative Teil der Analyse ist bei diesem Algorithmus auf die Abarbeitung der komplexen Zuweisungen beschränkt. Dabei ist zu beachten, dass kein transitiver Abschluss für den Graphen berechnet wird. Das bedeutet, dass zum Herausfinden der Zielvariablen eines Zeigers v nicht nur die Basisliste des Knotens n_v abgefragt werden muss, sondern die Listen der Basiselemente von allen Knoten im Graph, die von n_v aus erreicht werden können. Diese Knoten werden mit einer Tiefensuche ermittelt. Bei der Abarbeitung der komplexen Zuweisungen wird nach der Art der Zuweisung unterschieden:

- `*x=y`
  Bei Zuweisungen dieser Form werden zunächst die Knoten aller Zielvariablen von n_x gesucht. Allen gefundenen Knoten wird eine Kante zum Knoten n_y hinzugefügt.
- `x=*y`
  Zunächst wird, sofern nicht bereits geschehen, dem Knoten n_x eine Kante zum Knoten n_{*y} hinzugefügt. Weiterhin sei Z die Menge aller Zielvariablen von `y`. Nun werden n_{*y} Kanten zu allen Knoten n_z, $z \in Z$ hinzugefügt. Der Zwischenknoten n_{*y} wird genutzt, um die Geschwindigkeit der Analyse zu steigern. So sind die Zielwerte von `y` für jede Iteration nur einmal zu bestimmen, auch wenn es mehrere Anweisungen gibt, die `y` auf der rechten Seite einer Zuweisung dereferenzieren.

Eine weitere Verbesserung liegt darin, die im Graphen entstandenen Kreise zu eliminieren. Dabei wird nicht explizit nach Kreisen im Graph gesucht, sondern dies geschieht beiläufig bei der Tiefensuche zur Ermittlung der Zielvariablen für einen Knoten. Der Algorithmus zur Tiefensuche benutzt einen Stack, wo der Pfad von dem Knoten, für den der Suchalgorithmus gestartet wurde, zum derzeit bearbeiteten Knoten gespeichert ist. Für alle vom gerade bearbeiteten Knoten ausgehenden Kanten wird geprüft, ob der jeweilige Zielknoten - hier `v` genannt - sich bereits auf dem Stack befindet. Ist dies der Fall, so bilden `v` und alle Knoten, die oberhalb von `v` auf dem Stack gespeichert sind, einen Kreis. Alle Knoten, die diesen Kreis formen, werden dann durch einen neuen Knoten ergänzt, der die Knoten des Kreises repräsentiert. In der Liste der Basiselemente dieses Knotens werden alle Basiselemente

der Knoten im Kreis gespeichert. Ebenso werden alle Kanten, die von Knoten im Kreis ausgehen und zu Knoten führen, die nicht Teil des Kreises sind so abgeändert, dass sie vom neu erzeugten Knoten ausgehen. Um nicht alle Kanten im Graphen zu den bisherigen Knoten abändern zu müssen, wird in den Knoten des Kreises ein Verweis auf den repräsentierenden Knoten gespeichert. Damit die Funktion zur Suche der Zielvariablen in Zukunft die zusammengefassten Knoten verwendet, wird vor der Bearbeitung eines Knotens geprüft, ob es einen Repräsentanten für diesen Knoten gibt. Ist dies der Fall, wird stattdessen der Repräsentant verwendet. Auch bei eventuell noch abzuarbeitenden komplexen Zuweisungen werden, sofern vorhanden, Repräsentanten verwendet.

2.2.2 Intra-/interprozedural

Eine intraprozedurale Analyse beschränkt sich darauf, jeweils nur den Rumpf einer einzelnen Funktion zu analysieren. Die Werte von Funktionsparametern und von globalen Variablen werden als unbekannt angenommen. Ebenso sind keine Informationen über aufgerufene Unterfunktionen bekannt. Daher müssen konservative Annahmen für die Aliasbeziehungen gemacht werden. So muss angenommen werden, dass

- alle Funktionsparameter P auf einen einzigen Speicherbereich S zeigen können
- alle globalen Variablen G auf diesen Speicherbereich S zeigen können
- jede Variable aus P, S oder G auf jede globale Variable zeigen kann.

In dieser Betrachtung schließt die Menge der globalen Variablen auch Variablen ein, die zwar lokal innerhalb einer Funktion, allerdings mit dem Attribut `static` deklariert sind. Diese Variablen behalten ihre Gültigkeit über den gesamten Programmablauf und können auch außerhalb der Funktion, in der sie deklariert sind, verwendet werden. Dies ist beispielsweise der Fall, wenn die Adresse einer als `static` deklarierten Variable in eine global deklarierte Variable kopiert wird.
Ähnlich konservative Annahmen müssen für aufgerufene Unterfunktionen gemacht werden. So kann die aufgerufene Funktion neben globalen Variablen den Wert aller Variablen V ändern, die durch Dereferenzierung der Argumente des Funktionsaufrufs erreicht werden können. So muss die Menge von Variablen V ähnlich konservativ gehandhabt werden wie globale Variablen und Funktionsparameter: jede Variable $v \in V$ kann auf jede Variable in V sowie auf jede globale Variable zeigen. Jede globale Variable kann auf alle Variablen aus V zeigen.
Es ist offensichtlich, dass diese Art der Analyse sehr ungenau ist [HP98]. Eine einfache Verbesserung diese Analyse ist, für jede Variable abzuspeichern, ob sie mindestens einmal im gesamten Programm mit dem Adressoperator `&` angesprochen wird. Ist dies nicht der Fall, so ist es unmöglich, dass diese Variable das Ziel einer *Points-to*-Beziehung wird. Das verbessert insbesondere die Handhabung von globalen Variablen [GLS01].

Eine Verbesserung der Genauigkeit stellt sich ein, wenn der Datenfluss zwischen einzelnen Funktionen analysiert wird. Zwar werden bei dieser Analyse weiterhin Funktionen einzeln mit den in Kapitel 2.2.1 genannten Algorithmen analysiert, jedoch werden Funktionsaufrufe im Zusammenhang mit den übergebenen Parametern betrachtet. Es ist für die Skalierbarkeit entscheidend, die einzelnen Programmfunktionen in der richtigen Reihenfolge zu analysieren, da eine Funktion einen Einfluss auf das Verhalten einer anderen Funktion - etwa durch das Verändern einer globalen Variablen - haben kann. Von gleicher Bedeutung ist, wie häufig eine Funktion analysiert wird, wenn sie an mehreren Stellen im Programm aufgerufen wird. Diese Thematik wird im folgenden Unterkapitel näher betrachtet.

2.2.3 Kontext-sensitiv/-insensitiv

Bei einer Kontext-insensitiven Analyse werden alle Funktionsaufrufe zu einem Kontext zusammengefügt. Das bedeutet, dass eine Funktion nach Möglichkeit nur einmal mit einer Kombination aller Parameter berechnet wird. Daraus folgt natürlich, dass für Funktionen Kombinationen von Parameterwerten berücksichtigt werden, die im Programm gar nicht auftreten. Der daraus entstehende Datenfluss wird auch als *unrealizable path* bezeichnet. Das folgende Beispiel verdeutlicht dieses Problem:

```
int *f(int *x) {
  return x;
}
void main() {
  int a, b, *p, *q;
  p=f(&a);
  q=f(&b);
}
```

Listing 2.5: Unrealizable paths

Offensichtlich wird in diesem Programm in Variable p ein Zeiger auf Variable a und in q ein Zeiger auf die Variable b gespeichert. Eine kontext-insensitive Aliasanalyse allerdings würde die Funktion f nur ein einziges mal analysieren und dabei annehmen, dass der Parameter x sowohl auf Variable a als auch auf b zeigen kann. Daraus würde gefolgert, dass sowohl p als auch q auf a und b zeigen können.

Um Auswirkungen einer Funktion auf eine andere Funktion zu berücksichtigen, iteriert der Algorithmus über alle Funktionen. Um während jedem Iterationsschritt eine Funktion nur einmal analysieren zu müssen, wird zunächst ein Funktionsaufrufgraph des zu analysierenden Programms erstellt, dessen Knoten die Funktionen repräsentieren. Dabei bildet die Funktion main einen ausgezeichneten Knoten, an dem jeder Iterationsschritt beginnt. Wird bei der Analyse eines Knotens ein Funktionsaufruf gefunden, so wird dieser Funktionsaufruf als Kontext für die aufgerufene Funktion gespeichert und bei der Analyse dieser Funktion verwendet. Die Iteration über den Funktionsaufrufgraphen endet, wenn die Aliasbeziehungen einen Fixpunkt erreichen.

Insbesondere beim Umgang mit globalen Variablen zeigt eine *kontext-insensitive* Analyse Verbesserungen zu einer rein *intraprozeduralen* Analyse, doch das genannte Problem der *unrealizable paths* zeigt, dass es noch Verbesserungspotenzial gibt. Dieses Potenzial wird von *kontext-sensitiven* Algorithmen ausgeschöpft. Kritisch für diese Algorithmen ist jedoch die Skalierbarkeit, da die Zahl der Funktionsaufrufe exponentiell mit der Anzahl der Funktionen wachsen kann. Davon betroffen wäre insbesondere ein Algorithmus, der einfach die *kontext-insensitive* Analysemethode so modifiziert, dass eine Funktion für jeden Funktionsaufruf separat analysiert wird. Ziel bei *kontext-sensitiven* Algorithmen ist daher, Analyseergebnisse von aufgerufenen Funktionen zwischenzuspeichern und nach Möglichkeit weiterzuverwenden.

Einen Ansatz zur Zwischenspeicherung stellen sogenannte *transfer functions* dar. Diese Funktionen bilden eine Menge von Eingabedaten - Funktionsparameter und globale Variablen - auf einen Ausgabedatensatz - für die aufrufende Funktion sichtbare Veränderungen der Aliasbeziehungen - ab. Transferfunktionen dieser Form würden natürlich nur geringe Überschneidungen - und damit Wiederverwendbarkeit - der Funktionsaufrufe erzeugen. Eine Verbesserungsmöglichkeit ist, Funktionsparameter durch abstrakte Parameter zu ersetzen, auf die die realen Parameter abgebildet werden [Wil97]; auch Zielwerten von den Parametern, die Zeiger darstellen, wird ein abstrakter Speicherbereich zugeordnet. So sind nicht die tatsächlichen Parameterwerte bzw. Variablen von Bedeutung, sondern nur die Aliasbeziehungen zwischen diesen Parametern und deren Zielvariablen. Eine weitere Verbesserungsmöglichkeit besteht darin, den Definitionsbereich der Transferfunktionen nur auf den wirklich relevanten Teil der Aliasbeziehungen zu beschränken. So ist etwa im Listing 2.6 beim ersten Aufruf von `f` der Alias zwischen den Zielvariablen der Parameter `p` und `q` nicht relevant, da er auf die aufgerufene Funktion keinen Einfluss hat:

```
int f(int *p, int *q) {
  return 0;
}

void main() {
  int a, b, c;
  c=f(&a, &a);
  c=f(&a, &b);
}
```

Listing 2.6: Beispiel für Transferfunktion

Dies führt dazu, dass beim zweiten Aufruf von `f` die beim ersten Aufruf erzeugte Transferfunktion benutzt werden kann. Die Feststellung, welche Aliasbeziehungen für eine Funktion relevant sind, wird erst während der Analyse der Funktion durchgeführt; so sind abstrakte Parameter bzw. Speicherbereiche, auf die nie durch Dereferenzierung zugegriffen wird, nicht Teil des Definitionsbereichs der Transferfunktion. Mögliche Zugriffe durch aufgerufene Unterfunktionen müssen allerdings be-

rücksichtigt werden. Die Leistungsfähigkeit dieser Analyseform hängt direkt von der Wiederverwendbarkeit der erzeugten Transferfunktionen ab. Im Worst-case ist es weiterhin erforderlich, eine Funktion für jeden Aufruf separat zu analysieren. Enthält ein Programm Rekursionen, so ist es sogar möglich, dass unendlich viele Kontexte für eine Funktion existieren. Eine Lösung besteht darin, die Anzahl der Transferfunktionen für eine Funktion zu begrenzen. So ist es möglich, anstatt für einen Funktionsaufruf eine neue Transferfunktion zu erzeugen, eine bereits bestehende Transferfunktion zu verwenden, die alle Aliasse dieses Funktionsaufrufs enthält. Diese Lösung führt zu korrekten Ergebnissen, verringert allerdings die Genauigkeit.

Eine andere Möglichkeit ist, für jede Funktion eine sogenannte *summary* zu erzeugen. Dabei handelt es sich um eine möglichst kleine Anzahl von Anweisungen, deren Ausführung alle Effekte dieser Funktion auf eine aufrufende Funktion herbeiführt [NKH04a]. Auch dieser Algorithmus iteriert über den Funktionsaufrufgraphen, der für diesen Algorithmus topologisch sortiert wird. Ein Iterationsschritt besteht hier allerdings aus zwei Phasen. Zunächst wird in einer *Bottom-up*-Phase, also ausgehend von den Funktionen, die keine weiteren Funktionsaufrufe besitzen, in topologischer Ordnung für jede Funktion eine *summary* erstellt. Zu dieser Gruppe von Funktionen gehören auch solche mit indirekten Funktionsaufrufen, für deren Funktionszeiger keine *Points-to*-Werte bekannt sind. Durch die topologische Sortierung wird sichergestellt, dass für alle aufgerufenen Unterfunktionen eine *summary* vorliegt. Diese *summary* wird anschießend in alle aufrufenden Funktionen an die Stelle des Funktionsaufrufs einkopiert. Dabei werden formale Parameter mit den Argumenten im Funktionsaufruf ersetzt. Der *Bottom-up*-Phase folgt eine *Top-down*-Phase, um die eigentlichen Aliasbeziehungen zu ermitteln. Da alle Abhängigkeiten zwischen Funktionen nun in die jeweils aufrufende Funktion kopiert wurden, kann hier ein Algorithmus für die intraprozedurale Analyse, allerdings mit einigen Änderungen versehen, verwendet werden. Diese Änderungen sind notwendig, denn Auswirkungen auf die aufrufende Funktion sollen nicht mehr herbeigeführt werden. Bei diesen Auswirkungen handelt es sich um Zuweisungen zu globalen Variablen, *Points-to*-Werten von globalen Variablen sowie *Points-to*-Werten von Parametern. Zusätzlich können Funktionsaufrufe ignoriert werden, da diese Aufrufe durch die jeweiligen *summaries* ersetzt wurden.
Erreicht die Menge der Aliasbeziehungen einen Fixpunkt ist die Analyse vollständig durchgeführt. Durch die Programmmodifikationen ist dies der Fall, wenn der Funktionsaufrufgraph sich nicht mehr ändert. Änderungen ergeben sich nur, wenn neue Funktionsaufrufe in diesen Graphen aufgenommen werden, was durch indirekte Funktionsaufrufe geschehen kann. So ist zu erwarten, dass dieser Algorithmus für eine große Zahl von Programmen mit sehr wenigen, oft sogar nur einer einzigen Iteration auskommt.
Auch für diesen Algorithmus sind rekursive Funktionen problematisch, da in diesem Fall Zyklen im Funktionsaufrufgraphen enthalten sind. Das führt dazu, dass keine topologische Sortierung durchgeführt werden kann. Eine Lösung ist, alle

Funktionen in einem Zyklus zu einer einzigen Funktion zusammenzufassen. Dadurch werden Funktionen, die Teil von einem rekursiven Zyklus sind, kontextinsensitiv behandelt.

2.2.4 Flow-sensitive/-insensitive

Dieses Merkmal unterscheidet, ob der Kontrollfluss innerhalb einer Funktion bei der Analyse betrachtet wird. Bei einer nicht flusssensitiven Analyse werden Aliasbeziehungen nur einmal für die gesamte Funktion gespeichert. Es werden alle Aliasbeziehungen gespeichert, die bei einer beliebigen Ausführungsreihenfolge der Anweisungen auftreten können. Bei einer flusssensitiven Analyse dagegen werden die Aliasbeziehungen für jede Anweisung separat gespeichert. Dabei werden ähnlich wie bei einer Datenflussanalyse für jede Anweisung vier verschiedene Mengen von Aliasbeziehungen gespeichert:

- *input*: In dieser Menge werden alle vor Ausführung der Anweisung geltenden Aliasbeziehungen gespeichert.
- *generate*: Diese Menge enthält alle Aliasbeziehungen, die durch Ausführung dieser Instruktion neu erzeugt werden.
- *kill*: Diese Menge enthält Aliasbeziehungen aus der Menge *input*, die nach Ausführung der Anweisung nicht mehr existieren.
- *output*: Diese Menge enthält alle Aliasse, die nach Ausführung der Anweisung existieren. Es gilt $output = (input - kill) + generate$.

Die Menge *input* ergibt sich jeweils aus der Vereinigung der *output*-Mengen aller vorhergehenden Anweisungen. Da die Speicherung der vollständigen Aliasbeziehungen für jede Anweisung sehr speicherintensiv ist, werden häufig nicht für jede Anweisung die *input*- und *output*-Mengen gespeichert. Stattdessen werden diese Mengen nur einmal für jeden Basisblock[1] gespeichert. Um für eine beliebige Anweisung `S` in einem Basisblock das *input*- bzw. *output*-Set zu bestimmen, wird das *input*-Set des Basisblocks in Ausführungsreihenfolge um die *generate*-Sets bzw. *kill*-Sets der `S` vorhergehenden Anweisungen im Basisblock erweitert bzw. reduziert. Da der Analysealgorithmus die Anweisungen nur in Ausführungsreihenfolge abarbeitet, ergibt sich kein Nachteil bei der benötigten Rechenzeit für die Abarbeitung eines Basisblocks; die Abfrage der Aliasbeziehungen für eine Anweisung, die nicht den Beginn eines Basisblocks darstellt, dauert allerdings länger.

[1] Ein Basisblock ist eine Menge von Anweisungen mit linearem Kontrollfluss. Nur die erste Anweisung in einem Basisblock darf ein Sprungziel sein und nur die letzte Anweisung darf ein Sprungbefehl oder eine Verzweigung sein. Die Definition lässt sich auf maximale Basisblöcke einschränken; das sind Basisblöcke, die nicht Teil eines anderen Basisblocks sind.

Der Ablauf der flusssensitiven Analyse gestaltet sich so, dass ausgehend von der ersten Anweisung einer Funktion über alle Anweisungen bzw. Basisblöcke iteriert wird. Ein Iterationsschritt besteht dabei aus der Aktualisierung der *input*-Menge, der Berechnung der *generate*- und *kill*-Informationen und letztlich der *output*-Menge für alle Anweisungen bzw. Basisblöcke. Diese Iteration wird solange fortgeführt, bis ein Fixpunkt erreicht ist. Dieser Fixpunkt kann von der mit einzelnen Anweisungen arbeitenden Variante schneller erkannt werden als von der Basisblock-Variante. So ist es beispielsweise möglich, dass sich das *input*-Set für die erste Anweisung in einem Basisblock ändert, diese Änderung jedoch keine Auswirkung auf in deren *output*-Set hat. Während die mit Anweisungen arbeitende Variante der Analyse den Iterationsschritt abbrechen könnte, würde die mit Basisblöcken arbeitende Variante noch alle im Basisblock folgenden Anweisungen abarbeiten.

Der wesentliche Unterschied zur fluss-insensitiven Analyse besteht in der Berücksichtigung von *kill*-Informationen, auch *strong updates* genannt. In der nicht flusssensitiven Analyse kann diese Information nicht berücksichtigt werden, da eine beliebige Ausführungsreihenfolge der Anweisungen angenommen wird. Das schließt die Annahme ein, dass eine Anweisung, die eine bestimmte Aliasbeziehung zerstört, nicht ausgeführt wird. Wie Hind und Pioli [HP98] jedoch festgestellt haben, vergrößert die Berücksichtigung von *kill*-Informationen die Genauigkeit der ermittelten Aliasbeziehungen nicht signifikant. Ein Grund dafür kann sein, dass es in realen Programmen häufig einen verzweigten Kontrollfluss gibt. So ist es oft möglich, dass Anweisungen nur unter bestimmten Bedingungen ausgeführt werden. Dazu gehören if-then-, if-then-else-Konstrukte sowie Schleifen, die möglicherweise null mal durchlaufen werden. *Kill*-Informationen dieser Anweisungen bleiben dann auf den betreffenden Basisblock beschränkt. Listing 2.7 verdeutlicht dies an einem if-then-Konstrukt.

```
void main() {
  int i, *p, *q, *r, cond;
  p=&i;
  if(cond) {
    r=p=0;
  }
  q=p;
}
```

Listing 2.7: if-then-Konstrukt

Die Anweisung p=0 zerstört die Aliasbeziehung zwischen *p und i, so dass für die Variable r keine Aliasbeziehungen angenommen werden. Da der then-Teil jedoch nicht zwingend ausgeführt wird ist anzunehmen, dass es einen Alias zwischen *q und i gibt.

Als Nachteil der flusssensitiven Analyse bleibt die durch die Iteration höhere Laufzeit sowie der deutlich größere Bedarf an Speicherplatz aufgrund der Speicherung der Aliasbeziehungen für jede Anweisung bzw. jeden Basisblock.

2.2.5 Field-sensitive/-insensitive

Komplexe Datenstrukturen wie Vektoren und Variablen vom Typ einer struct-Deklaration bieten die Möglichkeit, Zeiger an mehreren Stellen innerhalb der Variable zu speichern. Gleichermaßen können Zeiger auf eine bestimmte Stelle in einer solchen Datenstruktur zeigen; dieser Sachverhalt ist in Listing 2.8 anhand einer struct dargestellt.

```
void main() {
  struct {
    int i, j, *p, *q;
  } s;
  int *u, *v, x, y;

  u=&s.i;
  v=&s.j;
  s.p=&x;
  s.q=&y;
}
```

Listing 2.8: Zeiger und komplexe Datenstrukturen

In diesem Beispiel sind die Felder i, j, p und q von s zu unterscheiden. So gibt es keinen Alias zwischen *u und *v; beide Zeigervariablen zeigen zwar auf die gleiche Variable, jedoch auf verschiedene Positionen innerhalb dieser Variablen. Auf ähnliche Weise lassen sich die Felder p und q von s unterscheiden. Diese Differenzierung wird bei einer *field-insensitiven* Analyse nicht vorgenommen. Stattdessen würde angenommen, dass sowohl u als auch v direkt auf s bzw. alle Elemente von s zeigen würden. Die in s enthaltenen Zeiger würden ebenfalls zusammengefasst. Dies hätte zur Folge, dass ein möglicher Alias zwischen *u und *v angenommen würde. Weiterhin würde angenommen, dass s.p auf y und s.q auf x zeigen kann. Vektoren werden in einer *field-insensitiven* Analyse vergleichbar gehandhabt; es wird nicht zwischen einzelnen Vektorelementen unterschieden. Für einen Zeiger auf ein bestimmtes Vektorelement wird angenommen, dass er auf jedes Vektorelement zeigen kann; weiterhin wird die Menge der möglichen Zielvariablen eines Vektorelements um die Zielvariablen aller anderen Vektorelemente erweitert.

Eine *field-sensitive* Analyse versucht dagegen, einzelne Felder einer struct und Vektorelemente getrennt zu betrachten. Dazu werden in einer solchen Analyse Adressberechnungen mit einbezogen und Indexausdrücke für Vektoren betrachtet. Da in diesen Berechnungen nicht nur Zeigervariablen vorkommen, wird auch eine allgemeine Datenflussanalyse vorgenommen. Dennoch ist es nicht immer möglich, die Ergebnisse von Adressberechnungen oder auch Indexausdrücken genau vorherzusagen. Daher muss die Menge der Speicherstellen, auf die durch einen Ausdruck zugegriffen werden kann, konservativ abgeschätzt werden. Da es keine Prüfung für die Einhaltung von Vektorgrenzen gibt, kann eine solche Menge unendlich viele Elemente besitzen.

Ein weiterer Aspekt einer *field-sensitiven* Analyse ist, dass plattformspezifische Eigenschaften einen Einfluss auf die Menge der Aliasbeziehungen haben können. Dies gilt etwa für das Beispiel 2.2 aus Kapitel 2.1. Zwar kann durch einen Zugriff auf b mit einem negativen Index auf den Vektor a zugegriffen werden, jedoch hängt es von der Ausrichtung der Vektoren a und b im Speicher ab, auf welches Element von a tatsächlich zugegriffen wird.
Eine Lösung der genannten Probleme liegt in der Verwendung von sogenannten *location-sets* [Wil97]. Dabei handelt es sich um ein Tripel $\{symbol, offset, stride\}$ mit der folgenden Bedeutung:

- *symbol* ist die Variable, auf die zugegriffen wird. Dabei handelt es sich um die Variable, oberhalb der in der Datenstruktur keine Adressbeziehungen bekannt sind. Im Beispiel 2.9 etwa würde für den Zugriff s.a[2] ein *location-set* mit dem Symbol s gebildet.

```
struct {
  int a[20];
  int b;
} s;
```

Listing 2.9: Schachtelung komplexer Datenstrukturen

- $offset$ gibt den Offset der zugegriffenen Speicherstelle in Bytes relativ zur Adresse der in *symbol* gespeicherten Datenstruktur an.

- $stride$ wird benutzt, um mehrdeutige Adressen abzubilden. Ein beliebiges Vielfaches dieser in Bytes spezifizierten Schrittweite kann auf die durch Symbol und Offset gebildete Adresse addiert werden. So bildet sich die Menge der möglichen Adressen für ein *location-set* folgendermaßen: $\&symbol + offset + n * stride, n \in Z$ (Vgl. Abbildung 2.2). Der Wert 0 für den Parameter Stride gibt folglich eine eindeutige Adresse an. Der Parameter $stride$ wird etwa für unbekannte Vektorindizes verwendet. Wird auf einen deklarierten Vektor int a[8] mit der unbekannten Indexvariable k zugegriffen (a[k]), so wird der Parameter $stride$ auf den Abstand zwischen zwei Vektorelementen von a gesetzt; bei einem Abstand von z.B. 4 etwa wäre das zugehörige *location-set* $(a, 0, 4)$. So kann eine unendlich große Menge an Vektorelementen, auf die durch einen Ausdruck möglicherweise zugegriffen werden kann, auf ein einziges *location-set* abbildet werden.

Ein feldsensitiver Analysealgorithmus arbeitet nun nicht mehr mit Variablen bzw. Ausdrücken, sondern den zugehörigen *location-sets*.
Das Problem des plattformabhängigen Speicherlayouts lässt sich durch den $offset$ einer Variablen bzw. des Felds einer struct lösen. Werden der Aliasanalyse die Größen primitiver Typen sowie deren Ausrichtung im Speicher übergeben, so lassen sich die Offsets der *location-sets* entsprechend berechnen.

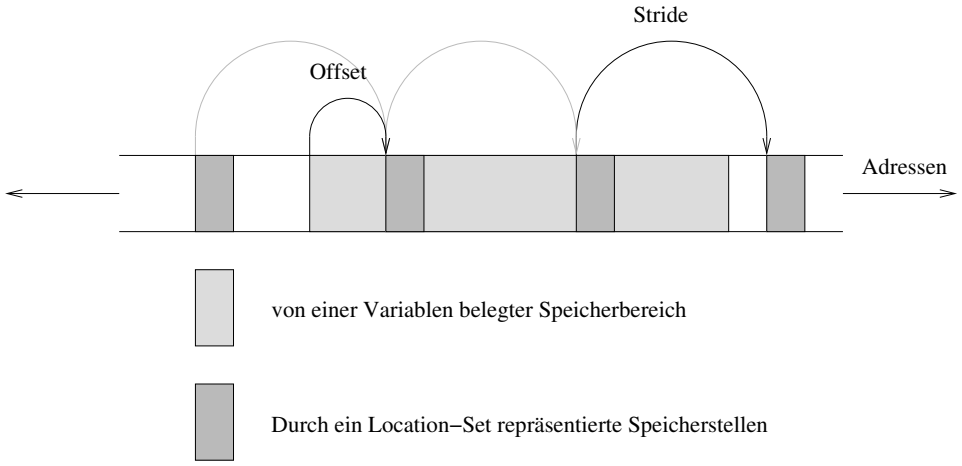


Abbildung 2.2: Location-Set

Wie stark die Genauigkeit der Aliasanalyse durch einen feldsensitiven Algorithmus zunimmt, hängt direkt davon ab, wie genau der Datenfluss anderer Variablen, die als Indexvariablen oder zur Adressberechnung benutzt werden, verfolgt wird. Je genauer die Analyse, desto größer die Schrittweite bzw. je geringer die Anzahl der mehrdeutigen *location-sets*. Ohne genaue Informationen über diese Variablen beschränkt sich der Vorteil auf die Differenzierung von Feldern in *struct*-Variablen, da der Zugriff auf diese Felder syntaktisch eindeutig ist (in Listing 2.9 etwa der Ausdruck `s.b`).

Die Skalierbarkeit nimmt jedoch mit zunehmender Genauigkeit ab; dies liegt nicht nur an der dann notwendigen komplexen Datenflussanalyse für alle Variablen, sondern auch daran, dass die Menge der *location-sets* die Menge der von einer *field-insensitiven* Analyse betrachteten Variablen deutlich übersteigen kann.

2.2.6 Highlevel-/lowlevel-Analyse

Innerhalb eines Compilers wird ein Programm in einer Zwischendarstellung gespeichert. Je nach Charakteristika dieser Zwischendarstellung spricht man von einer *Highlevel-* beziehungsweise *Lowlevel*-Darstellung [Muc97]. Eine *Highlevel*-Darstellung ähnelt der Sprachsyntax. Diese Darstellung beinhaltet Elemente wie Funktionen, Anweisungen, Schleifen, Ausdrücke und Variablen. Eine *Lowlevel*-Darstellung dagegen entspricht von der Struktur her viel eher einer Maschinensprache; Schleifen und Programmverzweigungen sind in Sprungbefehle zerlegt und hierarchische Ausdrücke in eine flache Struktur überführt. Die Anweisungen werden auf maschinenähnliche Anweisungen, häufig im Drei-Adress-Format, abgebildet. Diese Darstellung besitzt bereits große Ähnlichkeit mit Assemblercode und kann bereits maschinenabhängig sein; so können z.B. in dieser Darstellung bereits den Variablen Register zugewiesen werden.

Ein Compiler kann mehrere Zwischendarstellungen verwenden; so kann das Eingabeprogramm Schritt für Schritt von einer durch den Parser erzeugten *Highlevel*-Darstellung – eventuell mit weiteren Zwischendarstellungen (*Midlevel*-Darstellung) – in eine *Lowlevel*-Darstellung transformiert werden. In allen Zwischendarstellungen können Programmoptimierungen vorgenommen werden. Wünschenswert wäre natürlich, mit nur einer einzigen Aliasanalyse für alle Zwischendarstellungen die Aliasbeziehungen berechnen zu können; dies ist jedoch nicht möglich, da die einzelnen Optimierungsschritte das Programm soweit abändern können, dass die ursprünglich berechneten Aliasbeziehungen unnötig ungenau oder nicht mehr korrekt bzw. konservativ genug sind [GBT+05].

Es gibt aber noch weitere Kriterien, die eine *Highlevel*- und eine *Lowlevel*-Analyse unterscheiden:

- In einer *Highlevel*-Darstellung sind Typinformationen (und damit Größen) vollständig bekannt; in einer *lowelevel*-Darstellung ist dem nur teilweise so.
- Eine *Lowlevel*-Analyse kann bereits durch die verwendete Darstellung maschinenabhängig sein
- Binärprogramme können relativ einfach in eine *Lowlevel*-Zwischendarstellung transformiert werden. So bietet sich eine *Lowlevel*-Analyse an, wenn einige Programmteile nur als Binärdateien, z.B. in Form von Bibliotheken vorliegen.
- Die Ergebnisse einer *Highlevel*-Analyse können vielseitiger verwendet werden, da sie sich auf die Quelldateien beziehen. Wird das Programm nicht modifiziert, so behalten sie auch nach einer Überführung des Programms in eine *Lowlevel*-Darstellung ihre Gültigkeit.

2.3 Nutzung der Analyseergebnisse

Dieses Kapitel soll einen Überblick darüber geben, wie in einem Compiler auf die erkannten Aliasbeziehungen zugegriffen werden kann. Grundsätzlich lässt sich zwischen Zugriffsmöglichkeiten und Speicherung der Aliasbeziehungen innerhalb der Analyse unterscheiden; die einzelnen Varianten lassen sich ineinander überführen. Aliasse können auf folgende Arten gespeichert werden:

- **explizite Aliasse**
  Aliasse werden als Paar $< v_1, v_2 >$ beschrieben. v_1 und v_2 stellen Variablen bzw. Ausdrücke dar. In dieser Darstellung sind alle Aliasbeziehungen enthalten.
- **implizite Aliasse**
  Diese Darstellung ähnelt der expliziten Darstellung, jedoch wird nicht die

vollständige Menge von Aliassen gespeichert. Stattdessen wird eine minimale Menge von Aliassen gespeichert, aus der sich alle Aliasse ableiten lassen. Dafür wird ausgenutzt, dass Aliasse transitiv sind, d.h., aus den Aliasbeziehungen $< v_1, v_2 >$ und $< v_2, v_3 >$ lässt sich die Existenz der Beziehung $< v_1, v_3 >$ folgern.

- **Points-to-Darstellung**
  In dieser Darstellung werden die Zielvariablen von Zeigern betrachtet. Diese werden für jeden Zeiger separat gespeichert. Die *Points-to*-Menge eines Zeigers enthält dabei alle möglichen Zielvariablen, die dieser Zeiger enthalten kann.

Aus der *Points-to*-Darstellung lassen sich Aliasse folgendermaßen ermitteln. Dabei gilt für die Aliasbeziehung zwischen zwei Ausdrücken v_1 und v_2:

- Wenn weder v_1 noch v_2 Dereferenzierungen sind:
  In diesem Fall kann allenfalls ein statischer Alias vorliegen, d.h., v_1 und v_2 sind Symbolausdrücke mit der gleichen Variablen. Bei einer *field-sensitiven* Analyse kann dieser Fall auch bei Indexausdrücken für einen Vektor auftreten.

- Wenn - ohne Beschränkung der Allgemeingültigkeit - v_1 eine Dereferenzierung ist, v_2 jedoch nicht:
  Ein Alias $< *v_1, v_2 >$ existiert, wenn die *Points-to*-Menge der in $*v_1$ dereferenzierte Variable v_2 enthält.

- Wenn sowohl v_1 und v_2 Dereferenzierungen sind:
  Ein Alias $< *v_1, *v_2 >$ existiert, wenn sich die *Points-to*-Mengen der jeweiligen dereferenzierten Variablen überschneiden.

Als Beispiel sei angenommen, dass die Variable x im *Points-to*-Set von y ist. Daraus lässt sich ein Alias $< *y, x >$ erkennen.

Zur Überführung der Alias-Darstellung in eine *Points-to*-Darstellung werden nur Aliasse von Ausdrücken v_1 und v_2 betrachtet, wenn v_1 eine Dereferenzierung ist. Für die jeweils in v_1 dereferenzierte Variable bzw. den dereferenzierten Ausdruck wird v_2 in die *Points-to*-Menge eingetragen. Als Beispiel sei ein Alias $< *a, b >$ angenommen; daraus lässt sich eine *Points-to*-Beziehung von a mit der Zielvariablen b konstruieren.

Die explizite Aliasdarstellung zeichnet sich durch einen hohen Speicherbedarf aus, während in der impliziten Darstellung die Abfrage der Aliasbeziehungen eine linear mit der Anzahl der gespeicherten Aliasbeziehungen wachsende Laufzeit haben kann. Es sei angenommen, dass es die Aliasbeziehungen $< v_1, v_2 >$, $< v_2, v_3 >$, $\ldots$, $< v_{n-1}, v_n >$ gäbe. So existiert auch ein Alias zwischen $< v_1, v_n >$. Um diesen

Alias herauszufinden, sind alle gespeicherten Aliasse zu prüfen. Bei der *Points-to*-Darstellung hängt der benötigte Speicherplatz linear mit der Größe der *Points-to*-Sets zusammen. Die Laufzeit für die Abfrage, ob sich eine bestimmte Variable bzw. ein bestimmter Ausdruck im *Points-to*-Set eines Ausdrucks befindet, hängt lediglich von der für das Set verwendeten Datenstruktur ab; es wird eine einfache Suche ausgeführt. Bei einer Speicherung in balancierten Bäumen etwa ergibt sich Suchzeit, die logarithmisch mit der Größe des *Points-to*-Sets wächst.

Ferner wird zwischen *may-alias*- und *must-alias*-Beziehungen unterschieden; analoges gilt für die *Points-to*-Darstellung. Eine *must*-Beziehung ergibt sich, wenn ein Zeiger `p` eindeutig auf eine andere Variable zeigt. Dies ist der Fall, wenn das *Points-to*-Set von `p` nur einen Zielwert umfaßt, oder wenn es in der expliziten Darstellung nur eine einzige Aliasbeziehung mit `*p` gibt. Nullzeiger sind zu berücksichtigen. Bei einer *field-sensitiven* Analyse darf das dem Ausdruck im *Points-to*-Set zugehörige *location-set* nicht mehrdeutig sein, d.h., der Parameter $stride$ muss den Wert 0 haben.

2.4 Aliasanalyse in bestehenden Compilern

In diesem Kapitel soll für einige Compiler der jeweils implementierte Algorithmus zur Aliasanalyse klassifiziert werden. Die Informationen stammen aus [INT] [IBM] [DEC] [GCCa] [GCCb].

- **Intel C**
  Elemente einer Struktur werden nur als Quelle von *Points-to*-Beziehungen unterschieden.

- **DEC C**
  Dieser Compiler benutzt eine typbasierte Analyse, d.h., die *Points-to*-Werte für einen Zeiger werden auf die Variablen beschränkt, deren Typ dem jeweiligen Basistyp des Zeigers entspricht. Die Form der Analyse ist als flussinsenitiv zu klassifizieren. Felder von *struct*-Variablen werden unterschieden, einzelne Elemente von Vektoren nicht. Die Analyse arbeitet rein intraprozedural.

- **IBM XL-Compiler**
  Es wird ein typbasierter, interprozeduraler Analysealgorithmus verwendet.

- **gcc 3.3.1**
  Dieser Compiler verwendet eine typbasierte Aliasanalyse.

- **gcc 4.0**
  Dieser Compiler enthält einen intraprozeduralen Analysealgorithmus; interprozeduraler Datenfluss wird nur durch eine *address-taken*-Analyse unter-

sucht. struct-Variablen werden in einzelne Felder aufgespalten. Durch die interne SSA-Darstellung[2] wird der Kontrollfluss berücksichtigt.

Die Compiler mit einer nicht-typbasierten Aliasanalyse erlauben es dem Benutzer, die Analyse mit typbasierten Einschränkungen durchzuführen. Diese Option wird *strict-aliasing* genannt. Da *C* keine strenge Typprüfung beinhaltet ist es möglich, Programme zu schreiben, für die eine solche Analyse fehlerhafte Ergebnisse liefern kann. Allerdings entsprechen diese Programme nicht einem Standard wie z.B. *ANSI-C* oder *C99*.

C99 erlaubt es dem Programmierer, über das Schlüsselwort restricted für eine Variablendeklaration dem Compiler mitzuteilen, dass diese Variable keine Aliasbeziehungen hat.

[2] Static Single Assignment; siehe [Muc97]

Kapitel 3

Gegenstand der Arbeit

In diesem Kapitel wird der implementierte Analysealgorithmus detailliert beschrieben. Zunächst werden die Entwurfsentscheidungen erläutert und das Interface beschrieben, anschließend wird auf die einzelnen Aspekte der Analyse eingegangen.

3.1 Entwurfsentscheidungen

Wie in Kapitel 1.1 erwähnt, stellt die Codeerzeugung für eingebettete Systeme einen wichtigen Anwendungsfall für eine Aliasanalyse dar. Da es zwischen verschiedenen Systemen im Allgemeinen keine Hardwarekompatibilität gibt, soll die implementierte Analyse nicht auf eine bestimmte Plattform beschränkt sein. Sofern plattformspezifische Eigenschaften einen Einfluss auf die Aliasbeziehungen haben, sollen diese konfiguriert werden können. Aufgrund der geforderten Plattformunabhängigkeit wird für die Analyse eine *Highlevel*-Zwischendarstellung verwendet. Ein zusätzlicher Vorteil durch die Analyse einer *Highlevel*-Darstellung ist, dass die Aliasanalyse vor allen anderen im Compiler auftretenden Programmtransformationen ausgeführt werden kann; die so berechneten Ergebnisse beziehen sich direkt auf den Quellcode und können so auch außerhalb des Compilers, beispielsweise zur Programmvalidierung, genutzt werden.

Ein weiteres Ziel war, Aliasse mit möglichst hoher Genauigkeit erkennen zu können. Daher fiel die Entscheidung auf eine interprozedurale, kontext-sensitive Analyse. Dazu nutzt die implementierte Analyse die in Kapitel 2.2.3 vorgestellten *summaries*. Ein Vorteil gegenüber der Analyse mit *transfer functions* ist, dass eine *summary*-basierte Analyse Programme, die nicht vollständig im Quellcode vorliegen, besser analysieren kann. Natürlich werden Aliasse in beiden Varianten nur für die im Quelltext vorliegenden Programmteile berechnet, doch ist es vergleichsweise einfach, *summaries* für die nur in Binärform vorliegenden Programmteile zu entwickeln (verglichen mit der Definition von Transferfunktionen). Diese *summaries* lassen sich z.B. aus der Spezifikation einer nur in Binärform vorliegenden Bibliothek ableiten. Die entwickelten *summaries* können nun zum Programmquell-

text hinzugefügt werden; so können die Effekte auf aufrufende Funktionen abgeschätzt werden. Gäbe es keine Informationen über nur in Binärform vorliegende Programmteile, so müssten diese so konservativ wie in einer intraprozeduralen Analyse behandelt werden.

Die Aliasbeziehungen werden mit dem in Kapitel 2.2.1 vorgestellten Algorithmus von *Andersen* erzeugt; die Analyse ist also inklusionsbasiert. Weiterhin soll die Analyse *field-sensitive* sein, da viele Programme starken Gebrauch von *struct*-Variablen und Vektoren machen. Das gilt insbesondere für die im Bereich von eingebetteten Systemen häufig anzutreffenden Programme zur Signalverarbeitung. Sehr häufig werden hier Vektoren mit fester Größe verwendet. Um dieses Ziel zu erreichen, werden die in Kapitel 2.2.5 vorgestellten *location-sets* verwendet.

Da Flusssensitivität die Skalierbarkeit der Analyse stark verschlechtert, die Qualität der Ergebnisse aber nur wenig verbessert [HP98], wurde darauf zur Ermittlung der Aliasse innerhalb einer Funktion verzichtet. Verwendet wurde der Kontrollfluss allerdings für das Erzeugen einer *summary* für eine Funktion. In diesem Schritt der Analyse kann die Berücksichtigung des Kontrollflusses sowohl Laufzeit als auch Qualität der Analyse verbessern.

Unter den in Kapitel 2.3 vorgestellten Möglichkeiten zur Speicherung der Aliasbeziehungen erschien die Speicherung in *Points-to*-Form als guter Kompromiss. Aufgrund der in *C* möglichen Adressarithmetik sollen diese Werte nicht nur für Variablen, sondern für ganze Ausdrücke wie z.B. `*(a+1)` gespeichert werden. Zwar ließen sich die *Points-to*-Werte solcher Ausdrücke aus den *Points-to*-Beziehungen der Variablen ermitteln, doch würde die wiederholte Berechnung unnötig Rechenzeit kosten. Die gewählte Form entspricht der Zwischenspeicherung dieser Ergebnisse.

Grundsätzlich basiert der implementierte Algorithmus auf der *summary*-basierten Analyse [NKH04a], der jedoch in einer Reihe von Punkten verbessert wurde:

- In der entwickelten Analyse werden die *summaries* unter Berücksichtigung des Kontrollflusses berechnet. So kann der Datenfluss innerhalb einer Funktion mit höherer Genauigkeit zurückverfolgt werden. Dies führt zu genaueren Ergebnissen, da ohne Kontrollflussinformationen Zuweisungsketten in einer Funktion berücksichtigt werden, die in dieser Form nie ausgeführt werden können. In der flusssensitiven Analyse werden diese Zuweisungen nicht in die *summary* aufgenommen. Da die *summaries* dadurch auch eine geringere Größe haben, verbessert sich auch die Skalierbarkeit des Algorithmus, schließlich können Zuweisungen aus *summaries* im Funktionsaufrufgraphen von den Blättern bis zur `main`-Methode durchgereicht werden; ein Beispiel dafür wäre eine Zuweisung zu einer globalen Variablen.

- Die bisherige Analyse ist nur eingeschränkt als *field-sensitive* zu klassifizieren. Einzelne Felder von Vektoren werden nicht unterschieden; für struct-Variablen werden zwar einzelne Felder unterschieden, jedoch kann Adressarithmetik nicht berücksichtigt werden, da der allgemeine Datenfluss nicht analysiert wird. Daher können einzelne Felder nicht mehr unterschieden werden, sobald ein Zeiger auf eine solche Datenstruktur erzeugt wird.

  Die implementierte Analyse dagegen ist vollständig *field-sensitive*; eine parallel zur Aliasanalyse durchgeführte Datenflussanalyse erlaubt es, Adressarithmetik in einem Programm zu verfolgen und so Zugriffe auf komplexe Datenstrukturen möglichst genau zu bestimmen.

- Die Analyse von Heap-Variablen wurde verbessert. Der ursprüngliche Algorithmus erzeugt für jeden Aufruf von malloc eine globale Variable, die alle Heapblöcke repräsentieren soll, die mit diesem Aufruf erzeugt werden. Damit werden von der Analyse viele Aliasse zwischen einzelnen Heap-Objekten angenommen, die es nicht gibt. Die entwickelte Analyse nimmt einen solchen Alias nur an, wenn auch der Aufrufpfad, durch den der malloc-Aufruf erreicht wurde, identisch ist. Damit werden einzelne Heap-Objekte stärker differenziert.

3.2 Interface zum ICD-C

Beim *ICD-C*-Compilerframework handelt es sich um ein Compiler-Frontend, das eine *Highlevel*-Zwischendarstellung von *C*-Programmen zur Verfügung stellt. Es ist in *C++* implementiert. Die Konstrukte der Sprache *C* werden dabei auf die folgenden Klassen - einschließlich deren Unterklassen - abgebildet.

- **IR_SymbolTable:**
  Diese Klasse stellt eine Symboltabelle dar. Dabei kann es sich sowohl um eine globale als auch um eine lokale Tabelle innerhalb einer Funktion handeln.

- **IR_Type:**
  Durch diese Klasse kann ein Variablentyp dargestellt werden.

- **IR_Symbol:**
  Eine Instanz dieser Klasse stellt ein Symbol dar.

- **IR_Exp:**
  Diese Klasse stellt Ausdrücke dar; für die verschiedenen in *C* vorhandenen Ausdrücke gibt es jeweils Unterklassen, wie z.B. IR_SymbolExp, die ein Vorkommen eines Symbols im Programmquelltext darstellt. Geschachtelte Ausdrücke werden als Baum abgespeichert.

- **IR_Stmt:**
  Durch diese Klasse werden Anweisungen dargestellt. Bestimmte Anweisungstypen wiederum sind als Unterklassen von IR_Stmt ausgeführt. Zu diesen Unterklassen gehört IR_CompoundStmt. Diese Klasse stellt einen gesamten Block von Anweisungen dar, beispielsweise einen Funktionsrumpf oder einen Schleifenkörper. Es gibt Methoden in IR_CompoundStmt, um auf die einzelnen in diesem Block vorhandenen Anweisungen, die wiederum vom Typ IR_Stmt sind, zuzugreifen.

- **IR_Function:**
  Instanzen dieser Klasse stellen jeweils eine Funktion im Quelltext dar. Der Funktionsrumpf dieser Funktion wird jeweils in einem IR_CompoundStmt gespeichert. Der Funktionskopf wird durch ein IR_Symbol dargestellt, dessen Typ IR_FunctionType ist, eine Unterklasse von IR_Type.

- **IR_CompilationUnit:**
  Eine IR_CompilationUnit stellt einen Teil eines Programms dar, z.B. eine gesamte Eingabedatei.

- **IR:**
  Diese Klasse repräsentiert das gesamte Programm. Sie enthält eine Liste von IR_CompilationUnit-Instanzen.

- **IR_Configuration:**
  Diese Klasse nimmt Konfigurationsparameter für das *ICD-C* auf. Es gibt eine statische Instanz dieser Klasse.

Die Möglichkeit zur Abfrage der *Points-to*-Beziehungen ist in die Klassen IR_Exp und IR_Symbol integriert. Es sind Funktionen implementiert, die sowohl alle möglichen *Points-to*-Werte für einen Ausdruck beziehungsweise eine Variable zurückliefern können, als auch prüfen können, ob sich ein bestimmter Wert in der Menge der *Points-to*-Elemente befindet.
Der Klasse IR_Exp (Abbildung 3.1) wurden die folgenden Funktionen hinzugefügt:

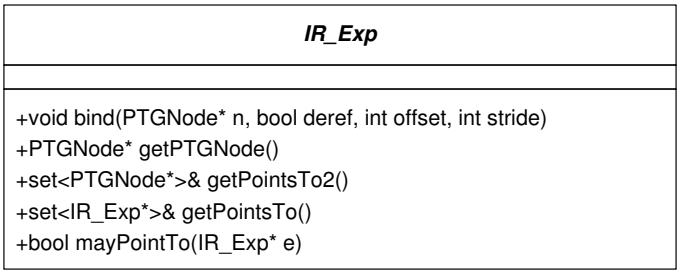


Abbildung 3.1: Klasse IR_Exp

- `set<IR_Exp*>& IR_Exp::getPointsTo()`
  Diese Funktion liefert für eine Instanz einer IR_Exp die Menge der *Points-to*-Werte zurück. Diese Werte werden als eine Menge von anderen Ausdrücken beschrieben.

- `set<PTGNode*>& IR_Exp::getPointsTo2()`
  Diese Funktion liefert ebenfalls die Menge aller *Points-to*-Werte zurück, dabei handelt es sich allerdings nicht um Ausdrücke. Stattdessen wird die Menge der im *Points-to*-Graph enthaltenen *location-sets* zurückgegeben, auf die der Ausdruck, für den diese Funktion aufgerufen wird, zeigt; diese *location-sets* werden durch die Klasse PTGNode dargestellt.

- **`bool`** `IR_Exp::mayPointTo(IR_Exp *e)`
  Mit dieser Funktion lässt sich prüfen, ob der Ausdruck e im *Points-to*-Set von dem Ausdruck ist, für den mayPointTo aufgerufen wird.

- `PTGNode *IR_Exp::getPTGNode()`
  Jeder Ausdruck wird mit einem *location-set* verknüpft; genauer ist dies in Kapitel 3.3.2 beschrieben. Diese Funktion liefert für einen Ausdruck einen Zeiger auf das jeweilige verknüpfte *location-set* zurück.

Die Funktionen getPointsTo und mayPointTo bieten sich an, um direkt mit den *Points-to*-Werten zu arbeiten. Die Funktionen getPTGNode und getPointsTo2 können dazu genutzt werden, um z.B. entsprechend der in Kapitel 2.3 vorgestellten Methode die *Points-to-* in eine explizite Aliasdarstellung zu transformieren.

Das Interface der Klasse IR_Symbol, die in Abbildung 3.2 dargestellt ist, hat einen ähnlichen Aufbau:

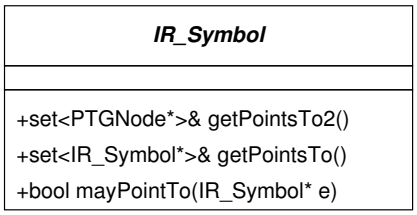


Abbildung 3.2: Klasse IR_Symbol

- `set<IR_Symbol*>& IR_Symbol::getPointsTo()`
  Diese Funktion liefert Zeiger auf alle möglichen Zielvariablen einer Instanz von IR_Symbol zurück. Da nicht bekannt ist, auf welche Adresse im Speicherbereich eines Symbols sich diese Funktion bezieht, arbeitet sie gemäß der in Kapitel 2.2.5 vorgestellten Klassifizierung nicht *field-sensitive*; es werden die *Points-to*-Werte von allen *location-sets* zurückgeliefert, die sich auf den Speicherbereich der Variable beziehen, für die diese Funktion aufgerufen wird.

- `set<PTGNode*>& IR_Symbol::getPointsTo2()`
  Diese Funktion verhält sich analog zur obigen Funktion, zurückgegeben wird die Menge von Zielvariablen aber nicht als Zeiger auf die entsprechenden IR_Symbol-Instanzen, sondern deren *location-sets*. Somit sind die Zielwerte sehr wohl *field-sensitive*; bei der Abfrage dieser Werte für ein IR_Symbol s allerdings werden die *Points-to*-Werte von allen *location-sets*, die sich auf den durch s definierten Speicherbereich beziehen, zusammengefasst.

- **`bool`** `IR_Symbol::mayPointTo(IR_Symbol *e)`
  Diese Funktion erlaubt es zu prüfen, ob ein bestimmtes Symbol im *Points-to*-Set eines anderen Symbols ist. Wie auch IR_Symbol::getPointsTo arbeitet diese Funktion vollständig *field-insensitive*.

Die *Points-to*-Beziehungen werden vor einer Abfrage neu berechnet, wenn dies notwendig ist. Das ist bei einer erstmaligen Abfrage sowie nach einer Manipulation der Zwischendarstellung der Fall. So gibt die Analyse immer ausreichend konservative Aliasbeziehungen aus, jedoch haben viele Programmmanipulationen keinen Einfluss auf die Aliasbeziehungen. So würden die Aliasse unnötig oft neu berechnet. Um die Neuberechnung der Aliasse unterbinden zu können, wurde die Klasse IR um die Funktion blockAliasRecomputation(bool b) erweitert. Wird diese Funktion mit dem Parameter true aufgerufen, werden Aliasse bei Programmänderungen nicht neu berechnet; ein Aufruf mit false führt zu einer automatischen Neuberechnung bei Änderungen der Zwischendarstellung. Nach einem Aufruf der Funktion blockAliasRecomputation und anschließender Veränderung der Zwischendarstellung können die errechneten Aliasbeziehungen ungültig sein, automatisch würden korrekte Aliasse aber frühestens mit einer weiteren Änderung der Zwischendarstellung berechnet. Daher gibt es eine Funktion, die die Neuberechnung von Aliassen erzwingt; dabei handelt es sich um die Funktion IR::recomputePoints-To. Es ist zu beachten, dass diese Funktion keinen Effekt hat, wenn die Aliasberechnung durch blockAliasRecomputation blockiert ist.

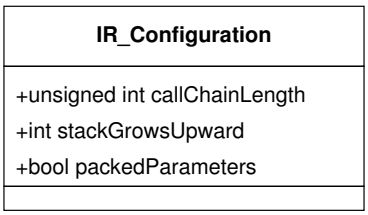


Abbildung 3.3: Erweiterung der Klasse IR_Configuration

Die Parameter, die zur Konfiguration der Aliasanalyse dienen, werden wie auch die restlichen Konfigurationsmöglichkeiten des *ICD-C* in der Klasse IR_Configuration untergebracht (Abbildung 3.3). Ein hinzugefügtes Attribut ist **`unsigned int`** `callChainLength`, über das die Genauigkeit, mit der Heap-Objekte analysiert wer-

den, eingestellt werden kann. Genauer wird darauf in Kapitel 3.3.8 eingegangen. Über die Attribute **int** stackGrowsUpward und **bool** packedParameters können maschinenspezifische Funktionsaufrufkonventionen definiert werden. Diese Informationen sind für die korrekte Bestimmung von *location-sets* (Kapitel 3.3.2) erforderlich. Je nach Wert von stackGrowsUpward wird angenommen, dass der Stack für die Übergabe von Funktionsparametern zu niedrigen bzw. hohen Adressen hin wächst. Ist das Attribut PackedParameters auf true gesetzt, so werden die Parameter anhand der Datentypgrößen ausgerichtet, sonst anhand der allgemeinen Ausrichtung im Speicher.
Die bereits innerhalb des *ICD-C* bereitgestellten Konfigurationsmöglichkeiten, die Größen von Datentypen und deren Ausrichtung im Speicher definieren, werden von der Analyse ebenfalls verwendet.

3.3 Detaillierte Beschreibung der Analyse

Abbildung 3.4 zeigt das Klassendiagramm der für die Analyse entwickelten bzw. erweiterten Klassen. Die Attribute und Methoden der einzelnen Klassen sind in den Abbildungen der folgenden Unterkapitel aufgeführt. Für die zum *ICD-C* gehörenden Klassen sind nur die Erweiterungen dargestellt. Die hinzugefügten Klassen haben dabei folgende Funktionen:

- **alloclib:**
  Diese Klasse stellt Funktionen zur Erzeugung von Heapblöcken zur Verfügung.
- **heapblock:**
  Durch Instanzen dieser Klasse können auf dem Heap liegende Speicherbereiche dargestellt werden. Es handelt sich um eine Unterklasse von IR_Symbol.
- **codeSimplifier:**
  Diese Klasse dient zur Umwandlung des Programmcodes, so dass dieser für eine *Andersen*-Analyse genutzt werden kann. Das Erzeugen von *summaries* basiert ebenfalls auf umgewandeltem Code. Der umgewandelte Code wird in Instanzen der Klasse simpleStmt gespeichert.
- **simpleStmt:**
  Durch diese Klasse werden "einfache" Anweisungen, wie sie durch den codeSimplifier erzeugt werden, dargestellt. Diese Anweisungen stellen im wesentlichen Zuweisungen von *location-sets* dar.
- **summary:**
  Diese Klasse dient zur Berechnung von *summaries* für eine Funktion; dies geschieht durch die Methode recreateSummary. Ebenso stellt diese Klasse die Funktionalität bereit, um eine *summary* an einen Funktionsaufruf anzupassen. Dies geschieht mit der Methode paste. Eine *summary* selbst wiederum besteht aus einer Liste mit Elementen der Klasse simpleStmt.

- **PTG:**
  Diese Klasse enthält die Funktionalität für einen *Points-to*-Graph. Dazu gehört insbesondere die Möglichkeit, die Auswirkungen eines simpleStmt zu der Menge der *Points-to*-Beziehungen hinzuzufügen. Dies geschieht über die Methode addAssignment. Um die möglichen *Points-to*-Werte für ein *location-set* herauszufinden, wird die Methode getLVals bereitgestellt.

- **PTGNode:**
  Instanzen dieser Klasse stellen einzelne *location-sets* dar. Gleichzeitig stellen diese Objekte Knoten im *Points-to*-Graph dar.

- **IR_SuperFunction:**
  Diese Unterklasse von IR_Function dient zur Bildung von rekursiven Funktionen.

Die Beziehungen zwischen den in Abbildung 3.4 dargestellten Klassen zeigen folgendes:

- Jede IR_Function enthält einen Verweis auf eine eigene Instanz von codeSimplifier und von summary.

- Sowohl codeSimplifier als auch summary enthalten Verweise zu Objekten vom Typ simpleStmt. Diese Anweisungen stellen einen umgewandelten Funktionskörper bzw. eine *summary* dar.

- Ein simpleStmt wiederum arbeit mit *location-sets*, die durch die Klasse PTGNode dargestellt werden. Die einzelnen Zuweisungstypen sind in Abschnitt 3.3.2 erklärt.

- *Location-sets* sind Instanzen von PTGNode. Sie stellen eine Variable oder einen Ausdruck im Programmquelltext dar und enthalten einen Zeiger auf ein IR_Symbol. Ferner können Ausdrücke (IR_Exp) und Objekte vom Typ PTGNode verbunden werden. Während jedem Ausdruck nur ein PTGNode zugeordnet werden kann, so kann ein PTGNode beliebig viele Ausdrücke im Programm darstellen.

- Auf diese Instanzen der Klasse PTGNode kann auch vom durch die Klasse PTG dargestellten *Points-to*-Graph aus zugegriffen werden. Es gibt genau eine Instanz von PTG. Die Klasse IR enthält einen Verweis auf diese Instanz.

- Die von der Klasse alloclib bereitgestellten Funktionen können in eine bereits bestehende IR_CompilationUnit eingefügt werden.

Grundsätzlich läuft die Analyse in mehreren Iterativen Schritten ab. In jedem Schritt werden zunächst Funktionen mit dem codeSimplifier umgewandelt. Aus diesem umgewandelten Code werden *summaries* erzeugt, die den jeweils für die aufrufende Funktion sichtbaren Datenfluss darstellen. Diese *summaries* werden anschließend

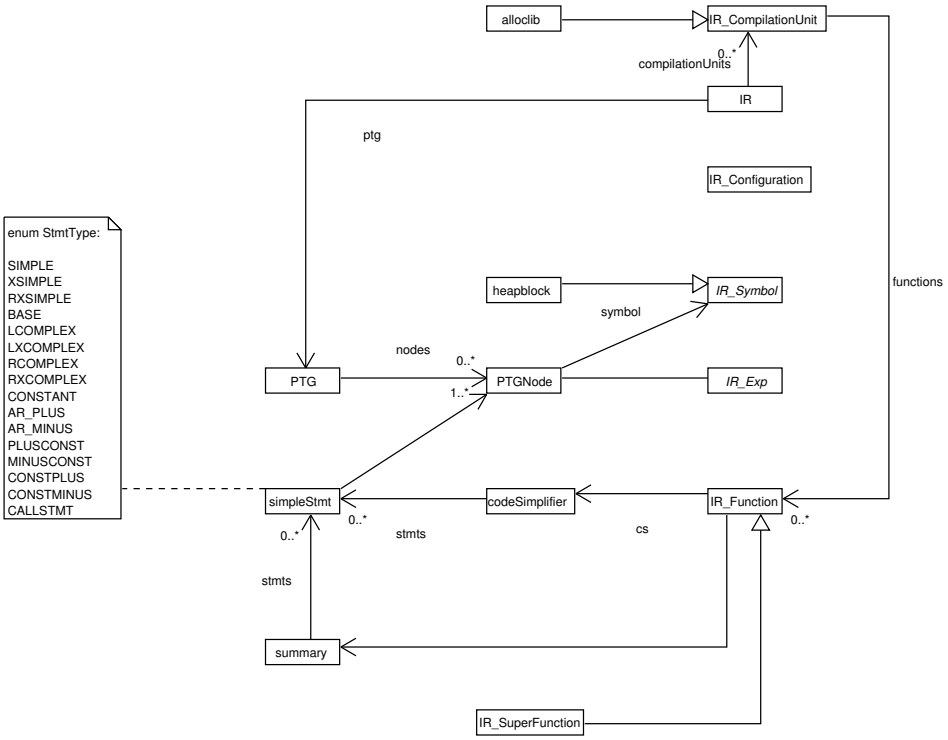


Abbildung 3.4: Klassendiagramm

an Stelle der Funktionsaufrufe eingefügt. Danach wird für jede Funktion separat die Menge der Aliasbeziehungen berechnet. Dieser iterative Schritt wird wiederholt, bis keine neuen Aliasbeziehungen mehr gefunden werden.

3.3.1 Location sets

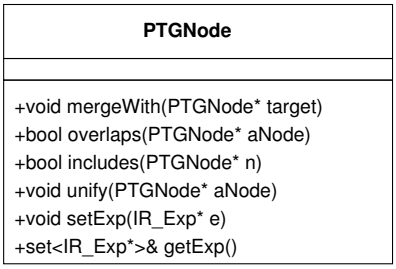


Abbildung 3.5: Klasse PTGNode

Die in Kapitel 2.2.5 vorgestellten *location-sets* lassen sich über Speicherbereich, Offset und Schrittweite beschreiben. So kann ein *location-set* als Tripel $< block, offset, stride >$ aufgefasst werden. Diese Sets entsprechen der in Abbildung 3.5 dargestellten Klasse PTGNode. Da diese Klasse auch die Knoten im *Points-to*-Graph darstellt, der nach der Methode von [HT01] aufgebaut wird, existiert in dieser Klasse ein weiteres Attribut, das speichert, ob es sich für einen Speicherbereich v um einen Knoten n_v bzw. n_{*v} handelt. Während ersterer Knotentyp einen Speicherbereich darstellt, handelt es sich bei letzterem nur um eine temporäre Variable, die die Zielwerte eines Zeigers zwischenspeichert.

Für einen Speicherbereich kann es verschiedene *location-sets* geben, die sich überlappen können; mittels der Methode PTGNode::overlaps kann ermittelt werden, ob sich zwei Sets überlappen. Durch diese Information kann herausgefunden werden, auf welche *location-sets* eine Zuweisung einen Einfluss hat, ob eine Zuweisung Teil von einem Zyklus ist oder sie kann genutzt werden, um den Datenfluss innerhalb einer Funktion zurückverfolgen zu können, was für die Erstellung einer *summary* benötigt wird. Die Funktion overlaps stützt sich darauf, dass sich zwei *location-sets* $l_1 =< b_1, o_1, s_1 >$ und $l_2 =< b_2, o_2, s_2 >$ genau dann überlappen, wenn gilt $b_1 = b_2$ und zwei natürliche Zahlen n_1 und n_2 existieren, so dass gilt $o_1 + n_1 * s_1 = o_2 + n_2 * s_2$. Eine Überschneidung wird nicht angenommen, wenn - ohne Beschränkung der Allgemeingültigkeit - l_1 einen Knoten n_{b_1}, l_2 aber einen Knoten n_{*b_1} darstellt. Grund dafür ist, dass n_{*b_1} keinen Speicherbereich darstellt, sondern nur die *Points-to*-Werte von n_{b_1} repräsentiert.
Da das Vorhalten vieler *location-sets* zu nur einem Speicherbereich negative Auswirkungen auf die Laufzeit der Analyse hat, wird versucht, deren Anzahl zu minimieren. Dies geschieht im Fall von mehreren sich überlappenden *location-sets*. Diese *location-sets* werden zusammengefügt. Ziel ist es, dass das zusammenge-

fügte *location-set* alle Speicherstellen der ursprünglichen *location-sets* enthält und dabei eine möglichst große Schrittweite besitzt. Grundsätzlich ist dieses Problem auf das Zusammenfügen von nur zwei *location-sets* $l_1 =< b_1, o_1, s_1 >$ und $l_2 =< b_2, o_2, s_2 >$ zurückzuführen. Dies ist einfach, wenn l_1 alle Speicherstellen von l_2 enthält, l_2 also einschließt. In diesem Fall entspricht das zusammengefügte *location-set* also l_1. Ob ein *location-set* l_1 das überlappende Set l_2 einschließt ist offensichtlich, wenn mindestens für ein *location-set* der Parameter $stride$ den Wert 0 hat. Ansonsten schließt von zwei sich überlappenden *location-sets* l_1 und l_2 das Set l_1 die Speicherstellen von l_2 genau dann ein, wenn die Schrittweite von l_2 ein Vielfaches der Schrittweite von l_1 ist. Komplizierter ist die Ermittlung eines optimalen zusammengefügten *location-sets* für l_1 und l_2, wenn sich diese Sets überlappen, aber nicht ein Set das andere einschließt. Um den Rechenaufwand in Grenzen zu halten, wird in diesem Fall nicht die maximale Genauigkeit angestrebt. Das neue *location-set* wird berechnet als $l_n =< b_1, o_1, gcd(gcd(s_1, s_2), |o_1 - o_2|) >$. *gcd* ermittelt den größten gemeinsamen Teiler der beiden Argumente. Da das neue Set alle bisherigen Speicherstellen abdecken soll, können die Parameter für den Speicherbereich und Offset unverändert bleiben. Die Schrittweite berechnet sich so, dass die Schrittweiten und Offsets von l_1 und l_2 getrennt betrachtet werden: der Ausdruck $x = gcd(s_1, s_2)$ berechnet eine Schrittweite x, die alle Speicherstellen von l_1 und l_2 einschließen würde, wenn beide *location-sets* den Offset 0 hätten. Erst der Ausdruck $y = gcd(x, |o_1 - o_2|)$ betrachtet die Offsets; die Schrittweite wird so sehr verkleinert, dass es ein $n \in Z$ gibt, so dass gilt $o_1 + n * y = o_2$. Da die ursprünglichen Schrittweiten s_1 und s_2 jeweils ein Vielfaches von x (und damit auch von y) sind, ist sichergestellt, dass das neu berechnete *location-set* alle Speicherstellen enthält, die durch die ursprünglichen *location-sets* repräsentiert wurden.

Das Zusammenfügen von *location-sets* wird nicht nur bei sich direkt überlappenden Speicherbereichen durchgeführt, sondern auch bei zyklischen Zuweisungen. In diesen Zyklen können mehrdeutige *location-sets* vorkommen. Beim Verschmelzen einer zyklischen Zuweisung zu nur einem einzigen Knoten in einem *Points-to*-Graph werden effektiv alle Speicherbereiche der im Zyklus enthaltenen Knoten zusammengefasst. Daher müssen auch hier die durch Offsets und Schrittweiten repräsentierten Speicherstellen konservativ abgeschätzt werden. Die Menge der Speicherstellen ist dabei durch die Mengen der einzelnen *location-sets* beschränkt; die Konstruktion mit der Schrittweite schließt bereits alle Speicherstellen ein, die durch ein mehrfaches Durchlaufen des Zyklus erreicht werden können. Das Verschmelzen von *location-sets* funktioniert daher äquivalent zum Zusammenfügen von sich überlappenden Sets.

Da die Klasse PTGNode auch Knoten im *Points-to*-Graph repräsentiert, werden auch *Points-to*-Beziehungen in dieser Klasse gespeichert. Da zur Speicherung dieser Beziehungen die in Kapitel 2.2.1 vorgestellte Methode von Heintze & Tardieu verwendet wird, werden Zuweisungen von Adressen und direkte Zuweisungen von Variablen getrennt gespeichert. Erstere Zuweisungen werden in der Liste baseEle-

ments gespeichert; wird die Adresse von einem *location-set* l_1 dem Set l_2 zugewiesen, so wird die Liste baseElements von l_2 um das Element l_1 erweitert. Direkte Zuweisungen von Variableninhalten werden in der Liste pointsTo abgespeichert. In beiden Listen werden dabei Zeiger auf die jeweiligen zugewiesenen *location-sets* gespeichert. Da die Analyse *field-sensitive* sein soll, müssen auch die zur Berechnung von Adressen verwendeten Konstanten gespeichert werden. Dafür wird das Attribut set<IR_ConstExp*> consts der Klasse PTGNode verwendet. Für nicht eindeutige Konstantenzuweisungen wird ähnlich nicht eindeutigen Adressen in einem Speicherbereich eine Schrittweite verwendet; dieser Wert wird im Attribut constStride abgelegt. Die Menge der möglichen Integerwerte, die eine Speicherstelle in einem location-set annehmen kann, kann also durch die Menge der zugewiesenen Konstanten sowie aller Werte, die sich ergeben, wenn beliebige Vielfache der Konstanten-Schrittweite zu diesen Werten addiert werden, abgeschätzt werden. Zusätzlich wird für jedes *location-set* die Größe des Datentyps gespeichert, unter dem auf die einzelnen Speicherstellen im Speicherbereich zugegriffen wird. Damit kann bei Zuweisungen festgestellt werden, wie groß der kopierte Speicherbereich ist.

3.3.2 Codeumwandlung

Um die Aliasbeziehungen entsprechend der in Kapitel 2.2.1 erläuterten *Andersen*-Analyse zu berechnen, sind komplexe Ausdrücke im Programm in mehrere einfache Ausdrücke umzuwandeln. Die Programmumwandlung geschieht jeweils für einen gesamten Funktionsrumpf. Dieser Funktionsrumpf wird in eine Liste von Instanzen der Klasse simpleStmt umgewandelt. Da bei der Umwandlung einige konservative Annahmen getroffen werden, etwa bei mehrdeutigen Indizes für Vektoren, kann es vorkommen, dass einige erzeugte Anweisungen wirkungslos sein können. Diese Anweisungen werden direkt nach der Umwandlungsphase eliminiert; Anweisungen, die wiederholt erstellt werden und auf einem Kontrollflusspfad liegen, werden zusammengefasst.
Für jede erzeugte simpleStmt-Zuweisung wird gespeichert, zu welcher ursprünglichen Anweisung vom Typ IR_Stmt sie gehören. So kann auch im umgewandelten Programmcode der Kontrollfluss verfolgt werden. Wenn Anweisungen zusammengefasst werden, so wird ein Intervall von Anweisungen gespeichert, das den zugehörigen Kontrollflusspfad einschließt. Um diese Bindung zu erstellen, enthält die Klasse simpleStmt die Methoden setMinStmt(IR_Stmt *s) und setMaxStmt(IR_Stmt *s), die mit den jeweiligen Intervallgrenzen aufgerufen werden. Bei einer Bindung zu einem einzigen IR_Stmt werden beide Funktionen mit identischem Argument aufgerufen.
Die erzeugten Anweisungen haben prinzipiell die Form von Zuweisungen; diese Zuweisungen beziehen sich auf *location-sets*. Im Allgemeinen wird für jeden Ausdruck im Programm (mindestens) eine “einfache” Anweisung erstellt; das Ziel-Set dieser Anweisung entspricht bezüglich der *Points-to*-Werte und möglichen Konstanten dem jeweiligen Ausdruck.

Folgende Arten von Anweisungen stehen dabei zur Verfügung. $l_1 \ldots l_n$ stellen *location-sets* dar:

- **SIMPLE:**
  Anweisungen dieser Art stellen eine direkte Zuweisung der Form $l_1 = l_2$ dar.

- **XSIMPLE:**
  Anweisungen dieser Art stellen ebenfalls eine Zuweisung dar; auch hier gibt es ein Ziel-*location-set* l_1 und ein Quell-*location-set* l_2, doch bezieht sich die Zuweisung nicht auf l_1, sondern auf andere Positionen im Speicherbereich von l_1. Um diese Position zu beschreiben, werden für Anweisungen dieser Form zusätzlich die Parameter $offset$ und $stride$ hinzugefügt. Während der Parameter $offset$ zum Offset von l_1 addiert wird, sind die Schrittweiten konservativer zu handhaben; die Schrittweite des effektiven Ziel-Sets ist der größte gemeinsame Teiler aus dem Parameter $stride$ und der Schrittweite von l_1. Dieser Ausdruck entspricht also $*(\&l_1 + offset + k * stride) = l_2$, dabei stellt k eine beliebige ganze Zahl dar.

- **RXSIMPLE:**
  Diese Anweisungen ähneln solchen vom Typ XSIMPLE; es wird jedoch bei diesem Typ die Quellvariable bzw. das Quell-Set um einen $offset$ bzw. eine neue Schrittweite verändert.

- **BASE:**
  Eine BASE-Anweisung stellt die Zuweisung einer Adresse dar: $l_1 = \&l_2$.

- **LCOMPLEX:**
  Diese Form von Zuweisungen stellt indirekte Schreibzugriffe der Form $*l_1 = l_2$ dar.

- **LXCOMPLEX:**
  Diese Anweisungen ähneln solchen vom Typ LCOMPLEX, jedoch wird hier die Speicheradresse um einen Offset und eine Schrittweite modifiziert; diese Anweisung stellt also $*(l_1 + offset + k * stride) = l_2$ dar; k ist wiederum eine ganze Zahl.

- **RCOMPLEX:**
  Anweisungen dieser Form sind indirekte Lesezugriffe der Art $l_1 = *l_2$.

- **RXCOMPLEX:**
  Diese Anweisungen verhalten sich zu RCOMPLEX wie LXCOMPLEX zu LCOMPLEX. Dargestellt wird $l_1 = *(l_2 + offset + k * stride)$.

- **CONSTANT:**
  Mit diesen Anweisungen werden Zuweisungen von Integerkonstanten im Programm dargestellt: $l_1 = k$; dabei stellt k eine Konstante dar.

- **AR_PLUS** und **AR_MINUS:**
  Diese Anweisungen können allgemeine Adressarithmetik - Additionen und Subtraktionen - darstellen: $l_1 = l_2 + l_3$ bzw. $l_1 = l_2 - l_3$.

- **PLUSCONST**, **MINUSCONST**, **CONSTPLUS** und **CONSTMINUS:**
  Mit diesen Anweisungstypen können Additionen und Subtraktionen von Konstanten dargestellt werden, z.B. $l_1 = l_2 + k$, $l_1 = l_2 - k$, $l_1 = k + l_2$ und $l_1 = k - l_2$; k soll eine Konstante darstellen. Häufig kann Adressarithmetik auch durch diese Anweisungen ausgedrückt werden.

- **CALLSTMT:**
  Durch diese Anweisungen können Funktionsaufrufe dargestellt werden. Auch bei Funktionen mit Rückgabetyp void wird eine Zielvariable angenommen. Die Anweisungsform lautet also $l_1 = l_2(l_3 \ldots l_n)$. Die *location-sets* l_3 bis l_n stellen die Funktionsargumente dar.

Die Funktionalität der Anweisungen vom Typ XSIMPLE, RXSIMPLE, LXCOMPLEX und RXCOMPLEX ließe sich zwar auf andere Anweisungen abbilden, jedoch würde dies das Einfügen zusätzlicher *location-sets* erzwingen; dies würde die Analysegeschwindigkeit verringern. Ähnliches gilt für die Anweisungen zur Addition und Subtraktion von Konstanten.
Die eigentliche Umwandlungsphase wird mit einem einzigen Durchlauf des Funktionskörpers durchgeführt; für jedes dort vorhandene IR_Stmt wird der vorhandene Ausdrucksbaum in *post-order* durchlaufen. Für jeden Ausdruck wird ein *location-set* erzeugt, das den Wert von diesem Ausdruck darstellt. Bezieht sich ein solcher Ausdruck auf keinen in einer IR_SymbolTable definierten Speicherbereich, so wird jeweils ein neuer Speicherbereich definiert. Diese für Ausdrücke erzeugten *location-sets* wiederum stellen die Operanden von Ausdrücken auf höherer Ebene im Ausdrucksbaum dar. Durch den *post-order*-Durchlauf beginnt die Umwandlung also bei Ausdrücken, die keine weiteren Teilausdrücke enthalten. Für alle anderen Ausdrücke sind die Teilausdrücke bereits umgewandelt und es existieren *location-sets*, die diese Ausdrücke darstellen. Für einen Ausdruck wird dann ein neues *location-set* erzeugt, das eine Verknüpfung der bestehenden Sets durch Anweisungen vom Typ simpleStmt darstellt. Dabei gibt es für die jeweiligen in *C* vorhandenen Ausdruckstypen die folgenden Regeln (die Ausdrücke sind nach den *ICD-C*-Klassen benannt):

- **IR_AssignExp:**
  Bei diesem Ausdruck handelt es sich um eine Zuweisung. Besonders zu betrachten sind Zuweisungen von komplexen Datentypen (Typ struct) sowie primitiven Typen, deren Größe diejenige eines Zeigers übersteigt. In diesen Fällen ist anzunehmen, dass Zeiger zugewiesen werden, die zwar in den jeweiligen Speicherbereichen gespeichert sind, jedoch an einer Stelle, die durch Addition eines Offsets zur Basisadresse dieses Bereichs erreicht werden kann. Daher wird bei Zuweisungen die Typgröße betrachtet und gegebe-

nenfalls auch weitere Speicherstellen kopiert; bei einer struct wird die Anordnung der einzelnen Komponenten betrachtet. Eine implizierte Typumwandlung wird wie eine explizite Umwandlung behandelt (siehe IR_UnaryExp). Weiterhin können die Ausdrücke auch arithmetische Komponenten enthalten, z.B. a+=b. Derartige Ausdrücke werden in die natürliche Form, in diesem Beispiel a=a+b, aufgeschlüsselt.

- **IR_BinaryExp:**
  Binäre Ausdrücke stellen arithmetische Operationen und Vergleiche dar. Während Addition und Subtraktion auf die Typen AR_PLUS und AR_MINUS von simpleStmt abgebildet werden, werden komplexere Berechnungen wie Multiplikation, Division, XOR etc. nicht berücksichtigt. Es wird angenommen, dass die Zielvariable jeden beliebigen Wert annehmen kann, d.h., der Wert *constStride* des erzeugten *location-sets* hat den Wert 1. Eine Optimierung ist, statt Anweisungen vom Typ AR_PLUS und AR_MINUS zu verwenden, auf solche vom Typ PLUSCONST, MINUSCONST, CONSTMINUS oder CONSTANT zurückzugreifen, wenn ein Operand eine Konstante darstellt. Der Fall, dass beide Operanden Konstanten sind, wird nicht berücksichtigt, da diese Werte bereits von einer *Constant-propagation*-Optimierung, die bereits vor der Berechnung der Aliasse durchgeführt werden kann, zusammengefasst werden können.

  Durch Vergleiche werden keine expliziten Zuweisungen ausgeführt, dennoch müssen diese konservativ abgeschätzt werden, wie folgendes Beispiel (Listing 3.1) zeigt:

```
int *i, j;
i=0;
while(i != &j)
  i++;
```

Listing 3.1: Zuweisung durch Vergleich

  Offensichtlich zeigt i nach Abarbeitung der while-Schleife auf die Variable j, obwohl es keine entsprechende Zuweisung gab. Daher müssen auch Vergleiche als Zuweisung betrachtet werden. Dabei können sowohl die Vergleichsoperatoren = und != als direkte Zuweisung aufgefasst werden. Andere Operatoren wie z.B. $<$ oder $>$ dagegen können nur vergleichsweise ungenau abgeschätzt werden. Hier kann die effektiv zugewiesene Variable in beliebiger Relation zur Vergleichsvariable stehen. Da jedoch nur Relationen innerhalb des Speicherbereichs einer Datenstruktur angenommen werden, sind die Zuweisungen auf alle Speicherstellen in diesem Bereich beschränkt. Die Menge dieser Stellen lässt sich durch ein *location-set* mit der Schrittweite 1 abschätzen.

- **IR_CallExp:**
  Diese Ausdrücke werden in ein CALLSTMT umgewandelt. Zu beachten ist die

in Kapitel 2.1 erwähnte, möglicherweise implizit vorhandene, Dereferenzierung von Funktionszeigern.

- **IR_ComponentAccessExp:**
  Bei einer IR_ComponentAccessExp kann es sich entweder um einen direkten Komponentenzugriff oder eine Dereferenzierung mit anschließendem Komponentenzugriff handeln. Letzterer Fall wird mit einer Anweisung vom Typ LXCOMPLEX bzw. RXCOMPLEX abgedeckt, je nach dem, ob der Zugriff auf der linken oder rechten Seite einer Zuweisung steht. Der zum dereferenzierten Zeiger addierte Offset entspricht dem Offset des Felds in der Zieldatenstruktur, die Schrittweite wird auf null gesetzt.
  Bei einem direkten Komponentenzugriff dagegen wird - auch für geschachtelte Strukturen - der Offset des Feldes relativ zur äußersten Struktur berechnet. Dieser Offset wird auf das *location-set*, das für den Ausdruck der äußersten Struktur ermittelt wurde, addiert.

- **IR_CompoundLiteralExp:**
  Bei diesen Ausdrücken handelt es sich um Zuweisungen zu komplexen Datenstrukturen (Vektoren und struct-Variablen). Diese Ausdrücke werden in Zuweisungen der einzelnen Elemente zerlegt.

- **IR_CondExp:**
  Es wird angenommen, dass dieser Ausdruck sowohl den Wert der Bedingung als auch beide möglichen Alternativwerte zurückliefern kann; dies dient wie bei einem Ausdruck vom Typ IR_BinaryExp der konservativen Abschätzung von Vergleichsoperationen.

- **IR_ConstExp:**
  Integerkonstanten werden durch eine CONSTANT-Anweisung abgedeckt. Andere primitive Datentypen, z.B. float, können nicht behandelt werden; es wird angenommen, dass jeder beliebige Wert zugewiesen wird. Für Zeichenketten wird ein neuer Speicherbereich definiert und ein Zeiger darauf zurückgegeben.

- **IR_IndexExp:**
  Ein Indexausdruck besteht aus einem Basis- und einem Indexausdruck. Handelt es sich bei letzterem um eine bekannte Konstante, so wird der Offset des *location-sets* des Basisausdrucks um diese Konstante, multipliziert mit der Typgröße des Basisausdrucks, erhöht. Ist der Indexwert unbekannt, so wird stattdessen eine Schrittweite gesetzt. Diese Schrittweite entspricht der Größe des Basistyps, wenn das *location-set* bisher keine *stride* hatte. Ist bereits eine Schrittweite vorhanden, so wird diese auf den größten gemeinsamen Teiler von Basistypgröße und bisheriger Schrittweite gesetzt. Für Vektoren variabler Länge wird immer eine Schrittweite gesetzt.

- **IR_InitListExp:**
  Diese Ausdrücke werden analog zu IR_CompoundLiteralExp behandelt.

- **IR_SymbolExp:**
  Für ein Symbol wird ein neues *location-set* erzeugt, für das sowohl Offset als auch Stride auf die Werte null gesetzt werden; es gibt allerdings folgende Ausnahmen:

  1. An Stelle von Funktionssymbolen kann deren Adresse gemeint sein (Kapitel 2.1). In diesem Fall wird ein weiteres *location-set* erzeugt, das auf das Set mit dem Funktionssymbol zeigt; dies geschieht durch eine Anweisung vom Type BASE.
  2. Ein nicht oder nicht vollständig indizierter Vektor verhält sich wie ein Zeiger, definiert ist aber nur der Speicher für den Vektor selbst. Daher wird für Symbole, deren Typ ein Vektor ist, im Ausdrucksbaum überprüft, wie viele Indexausdrücke vorhanden sind. Ist diese Zahl gleich den Dimensionen des Vektors, so wird das Symbol wie ein primitiver Typ behandelt. Ansonsten wird nur ein Zeiger auf den entsprechenden Speicherbereich zurückgeliefert. Dies geschieht analog zu Funktionssymbolen durch das Erzeugen eines temporären Speicherbereichs und das Einfügen einer BASE-Zuweisung, kann jedoch entsprechend der Zahl der zur vollständigen Indizierung "fehlenden" Indexausdrücke mehrfach wiederholt werden.

- **IR_UnaryExp:**
  Unäre Ausdrücke stellen eine ganze Klasse von Ausdrücken dar, deren Elemente im folgenden beschrieben werden:

  – **PLUS, MINUS, PRE-/POSTDEC, PRE-/POSTINC:**
    Diese Ausdrücke werden in binäre Form überführt, der Ausdruck $-a$ etwa in $0-a$ und entsprechend anderen binären Ausdrücken behandelt. Inkrement- und Dekrementausdrücke werden in die Form $a = a+1$ überführt. Pre- und Postoperatoren sind aufgrund der nicht vollständig genutzten Flusssensitivität als äquivalent zu betrachten.

  – **logisches NOT, bitweises NOT:**
    Der Ausdruck wird als unbekannt angenommen; er kann auf jede Stelle in jedem bekannten Speicherbereich zeigen und jeden Integerwert annehmen.

  – **Adressoperator:**
    Dieser Ausdruck wird durch ein simpleStmt vom Type BASE abgebildet.

  – **Dereferenzierung:**
    Dieser Operator wird durch eine LCOMPLEX- bzw. RCOMPLEX-Anweisung dargestellt.

- **sizeof:**
  Ist dem Compiler die Typgröße bekannt, wird sie als Integerwert betrachtet; ansonsten wird ein *location-set* zurückgeliefert, das eine Integervariable darstellt, die jeden beliebigen Wert annehmen kann.
- **cast:**
  Dieser Ausdruck ist von Bedeutung, wenn beide Typen nicht identische Größe haben. Je nach Speicherung von Daten im Hauptspeicher ist davon auszugehen, dass die LSB oder MSB abgeschnitten werden. Der resultierende Datenwert ist als unbekannt anzunehmen. Ist die Typgröße der Quellvariable größer als die der Zielvariablen, so wird dem Ziel-*location-set* jede Speicherstelle in der Quellvariablen bzw. im Quell-Set zugewiesen.

Die Verbindung zwischen Ausdrücken im Programm und *location-sets* wird ebenfalls in dieser Phase hergestellt. Dies geschieht durch die Methoden **void** PTGNode ::setExp(IR_Exp *e) und **void** IR_Exp::bind(PTGNode *n). Erstgenannte Methode fügt der von einer PTGNode-Instanz repräsentierten Ausdrücken e hinzu, während letztere Methode für einen Ausdruck definiert, von welchem *location-set* er repräsentiert wird.

Funktionsparameter

Funktionsparameter kommen im Funktionskörper als eigenständige Symbole vor. Dies würde bei der Annahme, es gäbe keine Beziehungen zwischen einzelnen Symbolen bezüglich der Speicheranordnung, möglicherweise zu fehlerhaften Resultaten führen. Das liegt daran, dass Funktionsparameter im Allgemeinen auf einem Stack gespeichert werden. Die genaue Form der Speicherung ist plattformspezifisch, jedoch weisen die Funktionsaufrufkonventionen vieler Plattformen eine große Menge von Gemeinsamkeiten auf. Üblich ist, die ersten *n* Parameter eines Funktionsaufrufs mittels Registerwerten zu übergeben, die restlichen Parameter werden auf einem Stack gespeichert. Für die in Registern übergebenen Parameter gibt es tatsächlich keine Beziehung zu anderen Speicherbereichen, für die auf dem Stack übergebenen Parameter allerdings sehr wohl. Daher führt die Annahme, alle Parameter werden auf dem Stack übergeben, zu korrekten Ergebnissen. Unbekannt ist jedoch, in welcher Anordnung die nicht in Registern übermittelten Parameter im Stackspeicher abgelegt werden. Zu unterscheiden ist, ob der letzte Parameter an der höchsten oder der niedrigsten Adresse abgelegt wird. Dies kann durch den Wert von IR_Configuration::stackGrowsUpward konfiguriert werden. Wird dieser Wert auf 1 gesetzt, so wird der letzte Parameter an der höchsten Adresse gespeichert, bei Wert 0 an der niedrigsten. Nun wird für den Stack ein neuer Speicherbereich definiert; für jeden Funktionsparameter wird ein neues *location-set* in diesem Speicherbereich definiert. Der jeweilige Offset wird aus der Position der Parameterliste errechnet. Da auf diese Speicherbereiche nur durch relative Adressierung zugegriffen werden kann, spielen die absoluten Adressen keine Rolle. Daher ist

auch hier die Annahme, alle Parameter würden auf dem Stack übergeben, sicher. Die einzelnen Offsets wiederum berechnen sich aus den Typgrößen bzw. dem normalen Anordnung der Variablen im Speicher; ersterer Wert wird verwendet, wenn der Wert IR_Configuration::packedParameters auf true gesetzt wird.
Das Beispiel in Listing 3.2 verdeutlicht die relative Adressierung über den Stack:

```
void f(int i, int j, int k) {
  int *x, *y;
  y = &j;
  x=y-1;
}
```

Listing 3.2: Stackadressierung

Je nach Richtung, in die der Stack wächst, kann x in diesem Beispiel nun entweder auf i oder k zeigen; es kann aber auch sein, dass die Variablen in Registern übergeben werden und es so keine gültige Zielvariable gibt.

Eine Besonderheit ergibt sich bei der Verwendung von Funktionen mit einer variablen Anzahl von Parametern. Eine solche Funktion wird definiert, indem die Parameterliste als letzten Eintrag um ein Auslassungszeichen (Ellipse) ... ergänzt wird, z.B. void f(int i, ...). Zunächst einmal werden diese Parameter wie explizit definierte Parameter behandelt. So ist es möglich, mittels Adressierung relativ zu anderen Parametern auf die variablen Parameter zuzugreifen. *C* definiert allerdings auch Makros, über die auf diese variablen Parameter zugegriffen werden kann. Diese Makros liefern eine Kopie des Parameterwertes zurück. Da der Rückgabewert dieser Makros zustandsabhängig sein kann, muss angenommen werden, dass jeder Parameterwert zurückgegeben werden kann. Ein solches zustandsabhängiges Makro stellt z.B. va_arg dar, mit dem die Liste der variablen Parameter durchlaufen werden kann; dazu wird ein interner Zähler definiert.

Da bei der Codeumwandlung die Aufrufkontexte noch unbekannt sind, wird für die Makros angenommen, dass alle variablen Parameter an einer einzigen Stelle im Stack gespeichert sind. Erst wenn ein Aufruf einer solchen Funktion analysiert wird, kann festgestellt werden, mit welchen Parametern diese Funktion aufgerufen wird; erst dann kann die Zuordnung von Argumenten zu Positionen auf dem Stack vorgenommen werden; dies ist notwendig, um Adressarithmetik mit Parametern zu ermöglichen. Allerdings wird im Zusammenhang mit der zusätzlichen Speicherung aller variablen Parameter an nur einer Stelle im Stack in der Analyse ein künstlicher Alias erzeugt. Durch das Einfügen einer zyklischen Zuweisung der beiden Positionen im Stack, an denen ein Parameter als gespeichert angenommen wird, werden jedoch die entsprechenden *location-sets* zusammengefügt. Daher ist dieser Alias nicht weiter zu berücksichtigen. Effektiv werden durch diese Vorhergehensweise die Parameter aus dem variablen Teil der Parameterliste nicht differenziert.

Nicht betrachtet wird die Möglichkeit, dass durch relative Adressierung zu Symbolen auf dem Stack auch auf Parameter von anderen Funktionen zugegriffen kann, die im Funktionsaufrufgraphen an höherer Stelle liegen. Diese Adressierungsmöglichkeit ist allerdings nicht sinnvoll verwendbar, da der Aufrufkontext als unbe-

kannt angenommen werden muss. Damit wäre auch der Bereich des Stacks, der keine lokalen Parameter enthält, als unbekannt anzunehmen.
Ferner werden alle return-Anweisungen umgewandelt in Zuweisungen zu einem neu erstellten Speicherbereich.

Entfernen von Anweisungen

Bei dieser Umwandlung kann es zu sich überlappenden *location-sets* kommen. Um diese nicht weiter berücksichtigen zu müssen, werden bereits in dieser Phase Vereinigungen von *location-sets* gebildet und die erzeugten Anweisungen entsprechend abgeändert. Daraus ergibt sich nun häufig, dass eine Anweisung keinen Effekt hat, da das Ziel-Set nur eine Teilmenge des Quell-Sets ist oder beide Sets identisch sind. Eine solche Anweisung kann entfernt werden. Eine weitere Möglichkeit, eine Anweisung zu entfernen ist, wenn es eine Kombination von Anweisungen $s_1 : \ l_1 = l_2$ und $s_2 : \ l_3 = l_4$ gibt und einen Kontrollflusspfad von s_1 zu s_2. s_2 kann entfernt werden, wenn l_1 und l_3 sowie l_2 und l_4 äquivalent sind. In einem solchen Fall wird die obere Intervallgrenze von s_1 auf s_2 ausgedehnt, wenn sich s_2 nicht bereits im Intervall von s_1 befindet.

3.3.3 Rekursion

Eine Rekursion entsteht, wenn sich eine Menge von Funktionen zyklisch aufruft. Ein Spezialfall ist, wenn sich eine Funktion selbst aufruft. Eine Rekursion ist im Funktionsaufrufgraphen als Kreis zu erkennen. Ein Kreis im Funktionsaufrufgraphen verhindert natürlich, dass dieser topologisch sortiert werden kann. Diese Eigenschaft ist allerdings, wie in Kapitel 2.2.3 erklärt, unabdingbar für die *summary*-gestützte Analyse. Folglich müssen diese Zyklen entfernt beziehungsweise ersetzt werden. Dies ist beispielsweise durch das Ersetzen von Rekursionen durch Iterationen möglich; außer in einigen Spezialfällen wie z.B. *tail-recursion*[1] ist dazu allerdings ein in Software verwalteter Stack erforderlich; dort können die Werte von lokalen Parametern zwischengespeichert werden. Ein ähnliches Modell wird in der Analyse implementiert, jedoch würde die Adressierung über den Stack ungenau sein. Das liegt daran, dass die Anzahl der auf dem Stack gespeicherten Elemente von der Rekursionstiefe abhängt; die Rekursionstiefe ist allerdings während der Programmanalyse nicht bekannt. Folglich sind die Speicherstellen, auf die die push- und pop-Operationen zugreifen nicht eindeutig. Die diesen Zugriffen zugehörigen *location-sets* hätten folglich also eine Schrittweite ungleich Null. Im günstigsten Fall wäre diese Schrittweite so bemessen, dass nur alle Instanzen einer lokalen Variablen in diesem *location-set* enthalten sind, d.h., es für jede lokale Variable ein eigenes *location-set* gibt und sich diese Sets nicht überlappen. Dieser Fall ist in Listing 3.3 und Abbildung 3.6 dargestellt. Eine derart genaue Analyse des Stacks ist jedoch in komplexen rekursiven Zyklen, wie z.B. in Abbildung 3.7 dargestellt,

[1] Eine *tail-recursion* bezeichnet den Fall, dass eine Rekursion durch einen Funktionsaufruf als letzte Anweisung in einer Funktion gestartet wird.

nicht möglich. In der Funktion f ergibt sich für die Variable a eine unterschiedliche Schrittweite für die Stackzugriffe, je nach dem ob die Funktion g oder h aufgerufen wird. Da sich die einzelnen *location-sets* überlappen, würden sie zu einem Set verschmolzen. Gleiches gilt für die *location-sets*, die die Speicherstellen der Variablen b darstellen; da sich die beiden resultierenden Sets ebenfalls überlappen, würden sie auch verschmolzen. So entsteht ein *location-set*, das Speicherstellen im Stack von sowohl a als auch b beschreibt. Im schlimmsten Fall ist es möglich, dass es für den Stack nur noch ein *location-set* gibt, dessen Schrittweite eins beträgt. Damit würden alle lokalen Variablen zu einer einzigen Speicherstelle zusammengefasst.

Die Genauigkeit ließe sich verbessern, wenn für jede lokale Variable innerhalb einer Funktion ein eigener Stack verwendet wird; so lassen sich immerhin verschiedene Symbole differenzieren. Da sich verschiedene Instanzen von lokalen Variablen jedoch weiterhin nicht auf einem Stack differenzieren lassen, kann dieser Stack zu einer einzigen Speicherstelle vereinfacht werden.
In der Implementierung wird dies soweit abstrahiert, dass ein lokales Symbol in einer Funktion nicht nur den Wert des aktuellen, sondern zusätzlich die Werte von allen bisherigen Aufrufen dieser Funktion enthält. Dies geschieht dadurch, dass der Aufruf von Funktionen, die ebenfalls ein Teil des rekursiven Zyklus sind, ersetzt wird durch ein Kopieren der Funktionsparameter in die lokalen Symbole; Im Beispiel in Listing 3.3 etwa würde der Rekursive Aufruf von f ersetzt durch eine Zuweisung i=j;.

```
void f(int i) {
  int j;
  /* ... */
  f(j);
  /* ... */
}
```

Listing 3.3: Rekursion

Da innerhalb rekursiver Zyklen auch die Erzeugung einer *summary* ohne Informationen über den Kontrollfluss durchgeführt wird, stellt diese Vorgehensweise eine konservative Abschätzung dar. Ein Rückgabewert einer aufgerufenen Funktion wird analog zu den Funktionsargumenten behandelt; eine Zuweisung des Ergebnisses eines Funktionsaufrufs wird durch eine Zuweisung von dem im codeSimplifier eingefügten Speicherbereich für return-Anweisungen ersetzt.

Besteht ein rekursiver Zyklus aus mehreren Funktionen, so werden die einzelnen Funktionskörper zusammengefügt. Dies geschieht mit bereits durch die Klasse codeSimplifier umgewandeltem Code. Das liegt daran, dass der Funktionsaufrufgraph durch indirekte Funktionsaufrufe zu Beginn der Analyse möglicherweise nicht vollständig bekannt ist. So ist es möglich, dass rekursive Zyklen erst während der Analyse erkannt werden. Durch die Nutzung von bereits umgewandeltem Code ergibt sich daher der Vorteil, dass eine Funktion nicht mehrfach umgewandelt werden muss. Die Analyse fügt für jeden erkannten rekursiven Zyklus eine Funktion

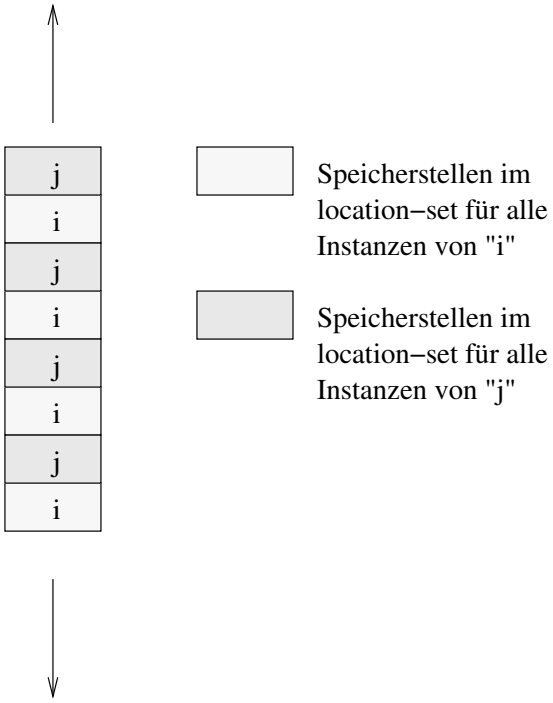


Abbildung 3.6: Stack bei einfacher Rekursion

vom Typ IR_SuperFunction in den Funktionsaufrufgraphen ein; die Funktionen, aus denen die neue Funktion gebildet wurde, werden nicht mehr betrachtet.

Diese Vorgehensweise bietet noch Spielraum für Verbesserungen. So wäre es möglich, statt den Stack in nur einer Speicherstelle zusammenzufassen, eine feste Anzahl von Speicherstellen zu definieren. Diese könnten zyklisch benutzt werden. Diese Speicherstellen können weiterhin den lokalen Variablen entsprechen; dann sind die im Zyklus vorhandenen Funktionen jedoch entsprechend oft zu kopieren. So könnte ein rekursiver Zyklus zu einem bestimmten Grad “abgerollt” werden, wie dies in Abbildung 3.8 dargestellt ist. Dort ist ein Teil des Funktionsaufrufgraphen dargestellt; es werden Kopien für die Funktionen f, g und h eingefügt. Das Verfahren kann beliebig oft ausgeführt werden, so dass ein Zyklus beliebig weit “abgerollt” werden kann. Das Kopieren der Funktionen ist insbesondere sinnvoll, wenn gleichzeitig eine Optimierung zur *function-specialization* durchgeführt werden soll. Darunter versteht man, dass die Menge der Aufrufkontexte für eine Funktion in mehrere Partitionen unterteilt wird. Eine Funktion wird dann so oft kopiert, dass es für jede Partition eine eigene Kopie gibt. Jetzt kann jede Kopie der Funktion unabhängig optimiert werden; da nur noch ein Teil der Aufrufkontexte zu betrachten ist, kann der Funktionskörper aggressiver optimiert werden.

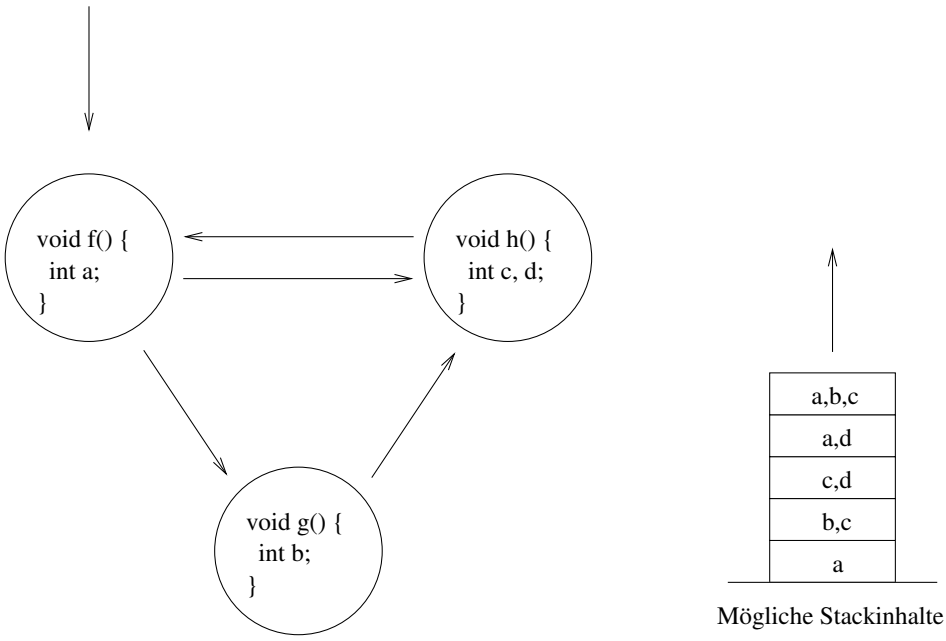


Abbildung 3.7: komplexer Rekursionszyklus

3.3.4 Iteration über den Procedure Call Graph

Der Ablauf der Analyse besteht aus einer Fixpunktiteration über den Funktionsaufrufgraphen. Im folgenden ist der Ablauf von einem Iterationsschritt erklärt. Zunächst wird eine Relation zwischen den einzelnen Funktionen definiert. Diese Relation $f_1 < f_2$ besagt, dass f_1 von f_2 aus aufgerufen werden kann. Indirekte Funktionsaufrufe werden nur berücksichtigt, sofern sie bereits bekannt sind. Mit dieser Relation wird zunächst eine Liste erstellt, die die Funktionen enthält, die von der Funktion main aus bzw. der mittels IR_Configuration::entryFunction spezifizierten Hauptfunktion des Programms aus erreicht werden können. Anschließend werden Zyklen in dieser Relation gesucht und jeweils durch eine IR_SuperFunction ersetzt. Die erzeugte Liste wird mit dem in [Weg] vorgestellten Algorithmus topologisch sortiert. Im nächsten Schritt wird geprüft, ob es innerhalb der zu analysierenden Programmteile Funktionsaufrufe gibt, die eine nicht im Quelltext vorliegende Funktion aufrufen. Diese Funktionsaufrufe werden entsprechend einer intraprozeduralen Analyse konservativ abgeschätzt. Anschließend beginnt die *Bottom-up*-Phase des Iterationsschrittes. Hier wird die sortierte Liste ausgehend von den Funktionen durchlaufen, die keine weiteren Funktionsaufrufe enthalten. Durch die topologische Sortierung ist gewährleistet, dass bei der der Analyse einer Funktion bereits alle aufgerufenen Funktionen analysiert worden sind. Werden die Funktionen zum ersten mal analysiert, wird die Codeumwandlung durchgeführt. Anschließend werden die *summaries* der bekannten Funktionsaufrufe eingefügt, dann für diese Funktion selbst eine *summary* berechnet. Eine Ausnahme bildet die Haupt-

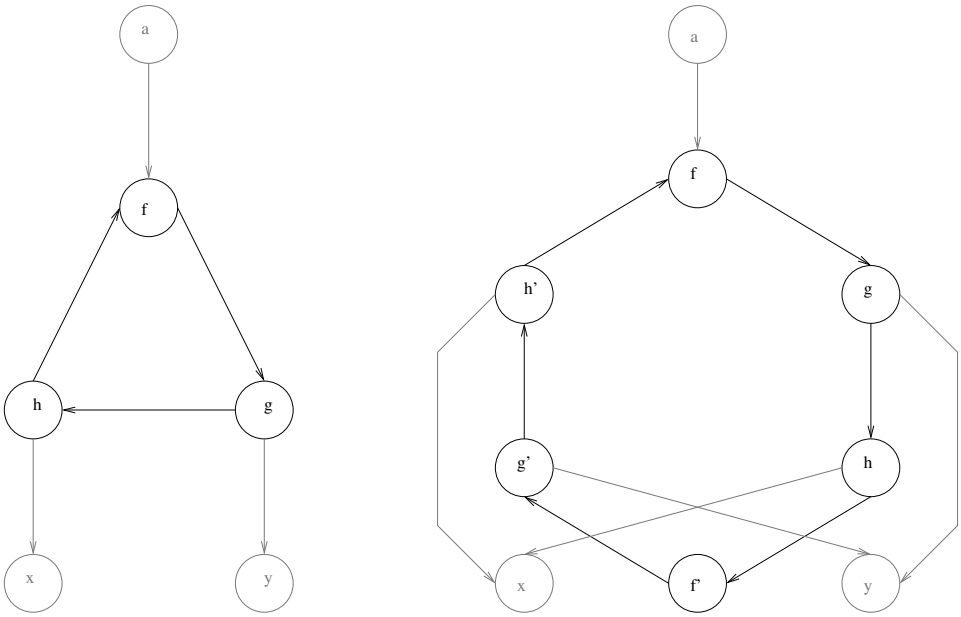


Abbildung 3.8: “Aufrollen” von einem rekursiven Zyklus

funktion; für diese Funktion ist es unnötig, eine *summary* zu berechnen, da es nur einen einzigen, bekannten Aufrufkontext gibt. Abbildung 3.9 zeigt das so modifizierte Programm. In dieser Abbildung stellt f' den Teil einer Funktion f dar, der Auswirkungen auf die aufrufende Funktion hat.

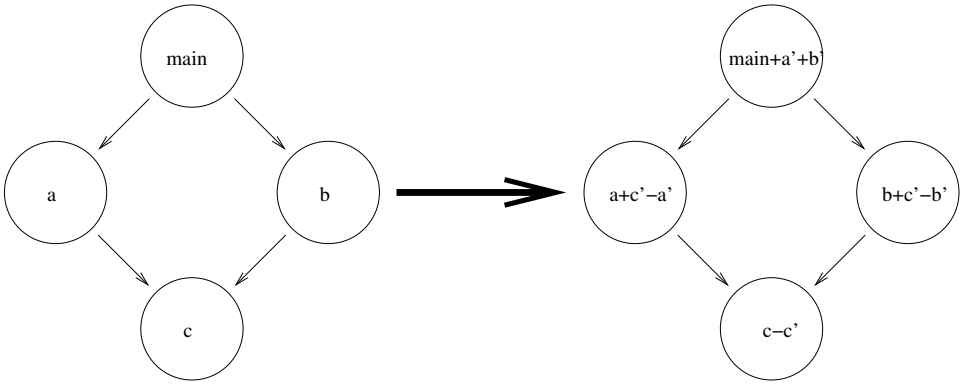


Abbildung 3.9: Verschieben von interprozeduralen Seiteneffekten in aufrufende Funktionen

Folgend auf die *Bottom-up*-Phase wird die topologisch sortierte Liste in der *Top-down*-Phase von der Hauptfunktion aus durchlaufen. Durch die Sortierung ist sichergestellt, dass in dieser Phase für eine Funktion alle Aufrufkontexte bekannt sind. In dieser Phase werden für jede Funktion intraprozedural die jeweiligen *Points-to*-Werte berechnet. Diese Berechnung wird nur für lokal deklarierte Variablen

durchgeführt, um keine für die aufrufende Funktion sichtbaren Aliasse zu erzeugen; diese werden ja bereits durch die eingefügte *summary* berechnet. Initialisierungen von globalen Variablen werden nur im ersten Iterationsschritt berücksichtigt.

Diese Iteration kann abgebrochen werden, wenn sich der Funktionsaufrufgraph nicht mehr ändert. Da Effekte von aufgerufenen Funktionen vollständig in den aufrufenden Funktionen bearbeitet werden, können sich die *Points-to*-Werte ohne eine Änderung des Funktionsaufrufgraphen nicht verändern. Ein weiteres Abbruchkriterium ist, wenn sich in der *Bottom-up*-Phase keine *summary* im Vergleich zum vorhergehenden Iterationsschritt geändert hat; diese Bedingung schließt ein, dass keine neuen Knoten zum Funktionsaufrufgraph hinzugefügt worden sind. So arbeitet die *Top-down*-Phase auf der gleichen Basis wie in der vorhergehenden Iteration. Dies würde wiederum zu identischen *Points-to*-Ergebnissen führen. Der Iterationsschritt kann also abgebrochen werden.

Weiterhin ist es möglich, dass sich in einer *Bottom-up*-Phase nicht für alle Funktionen die *summary* ändert. Dies führt ebenfalls dazu, dass die Ergebnisse in der *Top-down*-Phase sich für einige Funktionen nicht von der vorhergehenden Iteration unterscheiden. Daher ist es sinnvoll, diese Menge von Funktionen zu ermitteln und in der *Top-down*-Phase zu überspringen. Zur Bestimmung dieser Funktionen ist der in Ebenen aufgeteilte Funktionsaufrufgraph heranzuziehen; die Funktion main soll auf Ebene eins liegen, die aufgerufenen Unterfunktionen auf höheren Ebenen. Wenn jetzt die Ebene n die kleinste Ebene ist, auf der eine Funktion existiert, deren *summary* sich in der letzten *Bottom-up*-Phase geändert hat, so können durch das Einfügen der *summaries* nur Funktionen, die Funktionen auf Ebene n oder einer noch größeren Ebene aufrufen, verändert werden. Diese Funktionen liegen alle auf einer Ebene größer als m. Folglich bleiben Funktionen auf Ebenen kleiner als m unverändert, schließlich enthalten diese Funktionen keine Aufrufe von Unterfunktionen, deren *summary* sich geändert hat. Da diese Funktionen in der anschließenden *Top-down*-Phase auch als erstes analysiert werden, ist es auch nicht möglich, dass sich neue Aufrufkontexte ergeben. Daher ist es möglich, die *Top-down*-Phase auf die Funktionen ab Ebene m zu beschränken.

Auf ähnliche Weise kann auch eine *Top-down*-Phase abgebrochen werden. Ändern sich die Werte von globalen Variablen und die Werte von Funktionsparametern für eine Funktion f nicht gegenüber der vorhergehenden Iteration, so kann diese Funktion in der *Top-down*-Phase übersprungen werden, wenn folgende Bedingungen erfüllt sind:

- Der von f aus erreichbare Teil im Funktionsaufrufgraph hat sich, verglichen mit der vorhergehenden Iteration, nicht geändert.
- Für keine im Funktionsaufrufgraph von f aus erreichbare Funktion hat sich die *summary* in der *Bottom-up*-Phase des aktuellen Iterationsschrittes geän-

dert; dadurch ist sichergestellt, dass sich die Menge der Anweisungen, die die Funktion f darstellt, sich seit der letzten Iteration nicht geändert hat.

Sind diese Kriterien erfüllt, so können auch alle im Funktionsaufrufgraph nur von f aus erreichbare Funktionen in der *Top-down*-Phase übersprungen werden. Das liegt daran, dass, verglichen mit der vorhergehenden Iteration, identische Funktionen mit identischen Aufrufkontexten analysiert werden.

3.3.5 Erzeugen von Summaries

Summaries werden erstellt, um die Auswirkungen einer Funktion auf die aufrufende Funktion kapseln zu können. Ziel ist es, diese Auswirkungen direkt in der aufrufenden Funktion herbeizuführen, was durch das Ersetzen eines Funktionsaufrufs durch die *summary* dieser Funktion geschieht. Um eine gute Skalierbarkeit der Analyse zu gewährleisten, soll eine *summary* aus möglichst wenigen Anweisungen bestehen.
Das Erzeugen einer *summary* gliedert sich in mehrere iterative Phasen. Zuerst wird der Datenfluss innerhalb der Funktion analysiert. Ziel ist es, herauszufinden, welche Variablen auf nicht lokal definierte Speicherbereiche zeigen. Mit dieser Information kann die Menge der simpleStmt-Zuweisungen ermittelt werden, die einen Einfluss auf die aufrufende Funktion haben können ("kritische" Anweisungen). Im darauf folgenden Schritt wird von diesen Anweisungen aus der Datenfluss zurückverfolgt, bis eine nicht-lokale Variable erreicht wird. Dieser Datenfluss wird anschließend in die *summary* kopiert. Dabei wird versucht, die Anzahl der Anweisungen in der *summary* möglichst gering zu halten, was durch das Zusammenfassen von mehreren Anweisungen gelingt. So kann beispielsweise eine Menge von Anweisungen $l_2 = l_1$, $l_3 = l_2$ durch die Anweisung $l_3 = l_1$ dargestellt werden. Anschließend wird die *summary* durch das Löschen überflüssiger Anweisungen weiter vereinfacht.

Finden kritischer Anweisungen

Um Anweisungen innerhalb einer Funktion herauszufinden, die einen Einfluss auf die aufrufende Funktion haben können, werden zwei Abbildungen eingeführt:

- $holding : Variable \longrightarrow int$
- $opaque : Variable \longrightarrow bool$

Die Abbildung *holding* besagt, wie häufig eine Variable zu dereferenzieren ist, bis möglicherweise auf einen Speicherbereich zugegriffen wird, der nicht lokal definiert ist. Der Wert -1 gibt an, dass keine solche Variable erreicht werden kann. Da in dieser Phase nur jeweils eine einzelne Anweisung, nicht aber der vollständige Datenfluss betrachtet wird, können einzelne *location-sets*, die zu einer Variablen

gehören, nicht differenziert werden. So ist es beispielsweise möglich, dass ein Zeiger auf eine Speicherstelle zeigt, von der aus nur lokal definierte Speicherbereiche erreicht werden können, jedoch durch Addition von einem Offset auf einen Zeiger auf eine externe Variable zugegriffen werden kann. Das Beispiel aus Listing 3.4 zeigt einen solchen Fall auf: Das *location-set* $< p,0,0 >$ zeigt auf den lokal definierten Speicherbereich $< t,0,0 >$. $< t,0,0 >$ entspricht der Variablen t.buffer. Eine Definition der *holding*-Eigenschaft auf *location-sets* würde also annehmen, dass von $< p,0,0 >$ aus keine externe Variable erreicht werden kann, d.h., *holding* für dieses *location-set* den Wert -1 annehmen würde. Wie das Beispiel aber zeigt, kann durch Addition von einem Offset zu p auf t.l zugegriffen werden, woraus folgt, dass durch erneute Dereferenzierung auf die globale Variable g zugegriffen werden kann; der Wert für $holding(p,0,0)$ würde also falsch berechnet. Durch die Definition von *holding* auf ganzen Speicherbereichen dagegen wird $holding(t)$ auf 1 und entsprechend $holding(p)$ auf 2 gesetzt.

Effektiv wird durch die genannte Vorgehensweise die *holding*-Eigenschaft nicht *field-sensitive* ermittelt. Das Beispiel hat gezeigt, dass für eine Berechnung auf Ebene der *location-sets* für jede Anweisung der gesamte Datenfluss zu analysieren wäre. Die implementierte Analyse jedoch beschränkt sich aus Performancegründen auf die Betrachtung jeweils einer einzelnen Zuweisung.

```
int g;

void f() {
  struct {
    int buffer;
    int *l;
  } *p, t;
  t.l=&g;
  p=&t;

  /* Zuweisung zu globaler Variable */
  *(p->l) = 0;
}
```

Listing 3.4: Beispiel zur Notwendigkeit konservativer *holding*-Definition

Durch die Funktion *opaque* wird dargestellt, ob der Kontext, in dem eine Funktion aufgerufen wird, einen Einfluss auf die Aliasbeziehungen einer Variablen haben kann. Dies ist der Fall, wenn die Adresse dieser Variablen einer externen Variablen oder einer bereits von einer externen Variablen aus erreichbaren Variablen zugewiesen wird. Diese Eigenschaft wird ebenfalls nur für ganze Speicherbereiche, also Variablen, bestimmt. Das liegt darin begründet, dass durch Adressarithmetik auf den gesamten Speicherbereich zugegriffen werden kann, sobald nur eine einzige Adresse aus diesem Bereich bekannt ist. Da während der Erstellung einer *summary* nicht bekannt ist, mit welchen Argumenten bzw. Aliasbeziehungen der Argumente eine Funktion aufgerufen wird, wird der diese Variablen berühren-

de Datenfluss vollständig in der *summary* abgebildet. Dies ist erforderlich, da es durch Aliasse indirekte Zuweisungen zu einer solchen Variablen geben kann, die ohne Kenntnis des Aufrufkontextes nicht zu erkennen sind; in der *Bottom-up*-Phase, in der die *summaries* erzeugt werden, sind aber noch keine Kontexte bekannt. In Listing 3.5 ist ein Fall dargestellt, in dem ein solcher kontextabhängiger Datenfluss auftritt.

In diesem Beispiel würde die lokale Variable x der Funktion f als *opaque* klassifiziert und so in die *summary* übernommen. Die beiden Aufrufe in g verdeutlichen die Auswirkungen verschiedener Aliasbeziehungen der Funktionsargumente auf die Effekte von f: Im ersten Aufruf gibt es einen Alias zwischen *r und *s. Damit wird durch die Anweisung *r=&x auch *s=&x gesetzt. Da x im ersten Aufruf auf den Wert 1 gesetzt wird, nimmt auch **s diesen Wert an. In Funktion f wird dieser Wert zugewiesen zu *p, was einer Zuweisung zur Variable a der Funktion g entspricht. Im zweiten Aufruf gibt es keinen Alias zwischen *r und *s. Damit wird *s nicht der Wert &x zugewiesen und **s nimmt nicht innerhalb der Funktion f den Wert von x, in diesem Fall 2, an. Damit wird durch die Anweisung *p=**s nicht der Zielvariable von p - b in der Funktion g - der Wert von Parameter q zugewiesen.

```
void f(int *p, int q, int **r, int **s) {
  int x = q;
  *r=&x;
  *p=**s;
}

void g() {
  int a, b, *c, *d, *e

  /* 1. Aufruf von f */
  f(&a, 1, &c, &c);

  /* 2. Aufruf von f */
  f(&b, 2, &d, &e);
}
```

Listing 3.5: Vom Kontext anhängiger Datenfluss

Um die *opaque*-Eigenschaft bestimmen zu können, wird eine weitere Abbildung $opaqueLevel : Variable \longrightarrow int$ eingefügt. Dieser Wert beschreibt, ausgehend von einer indirekten Zuweisung zu einer Externen Variable durch $*u = v$, wie häufig der Adressoperator in einer Zuweisungskette verwendet werden muss, bis die Adresse einer internen Variable z der Speicherstelle *u zugewiesen wird. Bei einer Menge von Zuweisungen $*u = v$, $v = *w$, $w = \&x$ und $x = \&z$ etwa kann $*u$ die Adresse von z zugewiesen werden. Dabei wird durch die indirekte Zuweisung $*u = v$ der Wert von *opaqueLevel* für v auf 1 gesetzt. Würde v die Adresse einer lokalen Variablen zugewiesen, so würde diese als *opaque* gekennzeichnet. Durch die Zuweisung $v = *w$ des Beispiels dagegen gilt dies nur für die Zielvariablen von w, damit gilt $opaqueLevel(*w) = 1$ und $opaqueLevel(w) = 2$. Nur der Wert

für *w* wird gespeichert. *x* ist eine solche Zielvariable von *w* und die *opaqueLevel*-Abbildung hat für diese Variable den Wert 1, so dass *z opaque* wird. Nicht direkt bekannte Zielwerte für eine Variable wie *w* kann es nur geben, wenn sich Zuweisungen durch Aliasing der Funktionsparameter ergeben. In diesem Fall wäre aber bereits *w* als *opaque* gekennzeichnet. Da alle Variablen, die durch Dereferenzieren von als *opaque* gekennzeichneten Variablen erreichbar sind, ebenfalls als *opaque* gekennzeichnet werden, ist die Berechnung ausreichend konservativer Ergebnisse sichergestellt.

Zu Beginn der Analysephase wird der Wert von allen Funktionsparametern in der Abbildung *holding* auf den Wert eins gesetzt. Dies besagt, dass durch Dereferenzieren von Funktionsparametern auf Speicherbereiche, die nicht lokal definiert sind, zugegriffen wird. Ferner werden alle globalen Variablen als *opaque* gekennzeichnet. Das besagt, dass diese Variablen Aliasbeziehungen mit Funktionsparametern bzw. deren *Points-to*-Werten besitzen können. Die in Kapitel 3.3.2 erwähnte Variable für Rückgabewerte wird ebenfalls als *opaque* gekennzeichnet. Diese Variable ist zwar frei von Aliassen, allerdings werden durch Annahme der *opaque*-Eigenschaft alle Zuweisungen zu dieser Variablen in die *summary* aufgenommen. Ausgehend von diesen Abbildungen werden nun für die lokalen Variablen Werte für die Abbildungen *holding*, *opaqueLevel* und *opaque* berechnet. Dies geschieht mit einer Iteration über alle Anweisungen. Funktionsaufrufe werden jedoch nicht betrachtet, da für diese aufgerufenen Funktionen bereits *summaries* eingefügt sind. Die Iteration arbeitet mit einer Reihe von Regeln, die in der bereits in Kapitel 2.2.1 erwähnten Notation beschrieben werden. Definitionen von *holding* und *opaqueLevel* können den jeweiligen definierten Wert nur verringern, sofern dieser vorher größer als 1 war. Die Bedingung $holding(v)$ ist erfüllt, wenn gilt $holding(v) \neq -1$.

- $\frac{u=CONST}{holding(u)=1}$: Es wird angenommen, dass die u zugewiese Konstante der absoluten Adresse einer externen Variablen entspricht. Daher wird $holding(u)$ konservativ mit dem Welt eins abgeschätzt.

- $\frac{u=*v \wedge holding(v)}{holding(u)=max(1,holding(v)-1)}$: Werden einem *location-set u* die Ziel-Sets eines anderen *location-sets v* zugewiesen, über das durch *n* Dereferenzierungen auf eine nicht-lokale Variable zugegriffen werden kann, so genügt es, *u* nur $n-1$ mal zu dereferenzieren, um auf diese externe Variable zugreifen zu können. Zeigt *v* direkt auf eine externe Variable, so wird angenommen, dass die Zielvariable von *v* auf eine weitere externe Variable zeigt; daher wird in diesem Fall der Wert $holding(u)$ als 1 definiert. Kann auch durch wiederholtes Dereferenzieren von *v* keine externe Variable erreicht werden, so hat diese Anweisung keinen Einfluss auf die *holding*-Abbildung von *u*. Die gezeigte Form der Regel betrachtet nur Anweisungen vom Typ RCOMPLEX, Anweisungen vom Typ RXCOMPLEX werden aber identisch behandelt. Das liegt daran, dass die Änderung der Zieladresse in einer RXCOMPLEX-Anwei-

sung bei der Abfrage von *holding* nicht relevant ist.

- $\frac{u{=}v \wedge hodling(v)}{holding(u){=}holding(v)}$: Bei einer einfachen Zuweisung nimmt das Ziel-*location-set* die Eigenschaften der zugewiesenen Variablen an. Auch hier wird die *holding*-Eigenschaft nicht *field-sensitive* betrachtet. Daher werden Anweisungen vom Typ XSIMPLE und RXSIMPLE analog behandelt. Diese Regel wird auch auf Anweisungen vom Typ PLUSCONST, MINUSCONST, CONSTPLUS und CONSTMINUS angewendet. Die bei diesen Anweisungen zusätzlich vorhandene Konstante wird nicht berücksichtigt; sollte mit dieser arithmetischen Operation eine absolute Adresse berechnet werden, so wird im gesamten Ausdrucksbaum eine einzelne Konstante vorkommen, durch die der Wert *holding* der Zielvariablen immer auf den Wert 1 gesetzt wird.

- $\frac{u{=}v{+}w \wedge holding(v)}{holding(u){=}hodling(v)}$, $\frac{u{=}v{+}w \wedge holding(w)}{holding(u){=}holding(w)}$: Diese Regeln entsprechen der obigen, jedoch auf Anweisungen vom Typ AR_PLUS bezogen. AR_MINUS-Anweisungen werden identisch behandelt. Sind beide Quellvariablen als *holding* klassifiziert, wird der *holding*-Wert der Zielvariablen auf den niedrigeren Wert der beiden Quellvariablen gesetzt. Ist die *holding*-Eigenschaft für nur eine Quellvariable erfüllt, so wird die Anweisung wie eine einfache Zuweisung dieser Quellvariable betrachtet. Kann durch keine Quellvariable eine nicht-lokale Variable erreicht werden, so kann durch diese Anweisung auch von der Zielvariablen aus keine solche Variable erreicht werden.

- $\frac{u{=}\&v \wedge opaque(u)}{opaque(v)}$: Kann die Variable (bzw. ein *location-set* dieser Variable) u von einer externen Variable aus erreicht werden und wird u ein Zeiger auf eine andere Variable v zugewiesen, so kann auch v von dieser externen Variable aus erreicht werden. Folglich wird auch v *opaque*.

- $\frac{{*}u{=}v \wedge holding(u){=}1}{opaqueLevel(v){=}1}$: Wird v einer externen Variablen zugewiesen, so werden alle Variablen *opaque*, deren Adressen in v gespeichert sein können. Dies ist die Bedeutung des Wertes 1 von $opaqueLevel(v)$; auf eine in v gespeicherte Adresse kann von einer externen Variablen aus zugegriffen werden.

- $\frac{u{=}v \wedge opaqueLevel(u){>}0}{opaqueLevel(v){=}opaqueLevel(u)}$: Wenn einer externen Variablen die Zielvariablen von u zugewiesen werden, so muss angenommen werden, dass u diese Zielvariablen durch eine Zuweisung von v erhält, wenn das Programm eine Anweisung u=v enthält. Diese Regel gilt auch für Anweisungen vom Typ XSIMPLE, RXSIMPLE, PLUSCONST, MINUSCONST, CONSTPLUS und CONSTMINUS. Bei Anweisungen vom Typ AR_PLUS und AR_MINUS ergibt sich der Wert der *opaqueLevel*-Abbildung für beide Quellvariablen aus dem Wert von $opaqueLevel(u)$.

- $\frac{u{=}v \wedge opaque(u)}{opaqueLevel(v){=}1}$: Durch diese Zuweisung werden alle Zielvariablen von v durch andere Regeln ebenfalls als *opaque* klassifiziert.

- $\frac{u=\&v \wedge opaqueLevel(u)>1}{opaqueLevel(v)=opaqueLevel(u)-1}$: Sind die Variablen, die durch n-maliges Dereferenzieren von u erreicht werden *opaque*, so kann es sich dabei um die Variablen handeln, die durch $n-1$-maliges Dereferenzieren von v erreicht werden.

- $\frac{u=\&v \wedge opaqueLevel(u)=1}{opaque(v)}$: Dies ist ein Spezialfall der obrigen Regel; da die Zielvariablen von u einer externen Variable zugewiesen werden, muss v als *opaque* markiert werden, da es sich um eine Zielvariable von u handelt.

- $\frac{u=*v \wedge opaqueLevel(u)}{opaqueLevel(v)=opaqueLevel(u)+1}$: Die durch n-maliges Dereferenzien von u erreichbaren Variablen werden einer externen Variable zugewiesen. Damit können auch die durch $n+1$-maliges Dereferenzieren von v erreichbaren Variablen dieser externen Variable zugewiesen werden.

- $\frac{opaque(u)}{holding(u)=1}$: Kann eine Variable u von einer externen Variable aus erreicht werden, so muss angenommen werden, dass u auf weitere externe Variablen zeigen kann. Das deckt den Fall ab, dass u auch durch Aliasing der Funktionsparameter mit einer weiteren externen Variable bzw. deren Adresse beschrieben wird. In diesem Fall ist kein direkter Schreibzugriff auf u erkennbar. Daher wird generell angenommen, dass eine solche Variable auf eine externe Variable zeigen kann.

- $\frac{u=\&v \wedge holding(v)}{hodling(u)=holding(v)+1}$: u wird ein Zeiger auf v zugewiesen. Kann durch n Dereferenzierungen von v aus auf eine externe Variable zugegriffen werden, so kann anschließend von u aus durch maximal $n+1$ Dereferenzierungen auf diese Variable zugegriffen werden.

- $\frac{u=*v \wedge opaque(u)}{opaqueLevel(v)=2}$: Kann u von einer externen Variable aus erreicht werden, so gilt dies auch für die Zielvariablen von *v, d.h. es muss $opaqueLevel(*v)=1$ gelten. Dazu wird $opaqueLevel(v)$ als 2 definiert. Die Regeln für BASE-Zuweisungen definieren anschließend die *opaqeLevel*-Eigenschaft für die Zielvariablen von v. Diese Regel wird auch auf Anweisungen vom Typ RXCOMPLEX angewendet.

Diese Regeln werden auf alle Anweisungen im Funktionskörper angewendet; dazu wird der Funktionskörper solange durchlaufen, bis die *holding*- und *opaque*-Eigenschaften einen Fixpunkt erreicht haben.

Anschließend folgt ein weiterer Durchlauf über alle Anweisungen, mit dem eine Abbildung $critical : Anweisung \longrightarrow bool$ definiert wird. Diese Abbildung liefert für eine Anweisung den Wert *true* zurück, wenn diese Anweisung eine direkte Auswirkung auf die aufrufende Funktion haben kann. Das ist in den im folgenden aufgelisteten Fällen möglich. Hier gilt, dass eine *holding*-Bedingung erfüllt ist, wenn der Rückgabewert dieser Funktion 1 ist.

1. $\frac{*u=v \wedge holding(u)}{critical(*u=v)}$: Wenn der Zielvariable einer Variablen u, die direkt auf eine externe Variable zeigen kann, ein Wert zugewiesen wird, kann eine externe Variable verändert werden. Diese Änderung wäre über den Funktionsaufruf hinaus sichtbar. Diese Regel wird auch auf Anweisungen vom Typ LXCOMPLEX angewendet.

2. Auf der linken Seite einer Zuweisung steht ein *location-set*, das Speicherstellen einer Variable u beschreibt. Gilt die Eigenschaft $opaque(u)$, so wird für diese Anweisung der Wert von $critical$ auf $true$ gesetzt. Diese Regel wird auf alle Zuweisungstypen außer LCOMPLEX und LXCOMPLEX angewendet. Letztere Typen sind ausgenommen, da bei diesen Zuweisungen das auf der linken Seite der Anweisung stehende *location-set* nicht beschrieben wird.

Vereinfachen des Datenflusses

Durch die bisherigen Schritte sind alle Anweisungen markiert, die einen unmittelbaren Einfluss auf die aufrufende Funktion haben können. In der nun folgenden Phase wird versucht, von diesen "kritischen" Anweisungen ausgehend den Datenfluss in der Funktion bis zu Parametern oder globalen Variablen zurückzuverfolgen. Dies geschieht über eine Iteration über alle "kritischen" Anweisungen. Da nach Möglichkeit mehrere Anweisungen miteinander verknüpft werden sollen, wird in jeder der folgenden Regeln jeweils der gesamte Funktionsrumpf durchlaufen. Dabei werden aber nur die Anweisungen betrachtet, von denen aus es einen Kontrollflusspfad zur jeweiligen betrachteten "kritischen" Anweisung geben kann. Dies ist für eine Kombination von einer kritischen Anweisungen s_1 und einer Anweisung s_2 einfach festzustellen, wenn diese mit exakt einem IR_Stmt verknüpft sind (Kapitel 3.3.2). Beschreibt diese Verknüpfung jedoch ein ganzes Intervall von Anweisungen, so wird die Kombination von s_1 und s_2 berücksichtigt, wenn es einen Kontrollfluss von einem IR_Stmt aus dem Intervall von s_2 zu einer Anweisung aus dem Intervall von s_1 gibt. Dies geschieht durch die Betrachtung der unteren Intervallgrenze von s_2 und der oberen Grenze von s_1. Da ein Intervall bedeutet, dass eine Zuweisung an einer beliebigen Stelle in dem durch dieses Intervall definierten Programmteil auftreten kann, werden durch die Abschätzung mit den Intervallgrenzen alle Kombinationen von Anweisungen berücksichtigt.
Da für ein IR_Stmt mehrere Objekte vom Typ simpleStmt erzeugt werden können und die Ausführungsreihenfolge der einzelnen Ausdrücke einer einzelnen Anweisung im Quelltext als unbekannt angenommen wird, werden auch Paare von Anweisungen s_1 und s_2 betrachtet, die mit dem gleichen IR_Stmt verknüpft sind.
Weiterhin ist für neu erzeugte Anweisungen, die eine Kombination der jeweiligen bisherigen Anweisungen darstellen, die Einordnung im Kontrollfluss zu bestimmen. Diese ergibt sich direkt aus der Position beziehungsweise dem Intervall von s_2.
Grundsätzlich arbeiten die folgenden Regeln so, dass beim Zurückverfolgen von den ursprünglichen kritischen Anweisungen aus der Datenfluss kompaktiert wird;

dazu werden neue kritische Anweisungen erzeugt bzw. andere, bereits bestehende Anweisungen, als kritisch markiert. Anweisungen werden erst in die *summary* eingefügt, wenn eine Eingabevariable erreicht ist oder aber der sich aus diesen Anweisungen ergebende Datenfluss nicht weiter zusammengefasst werden kann. Die Regeln beziehen sich auf den Typ der "kritischen" Anweisung s_1 in der untersuchten Kombination s_1, s_2. $input(v)$ bedeutet, dass v eine Eingabevariable ist, $summary(s)$ bedeutet, dass s Teil der *summary* ist bzw. wird. Die *holding*-Eigenschaft gilt als erfüllt, wenn der Wert ungleich -1 ist. Aufgrund der Definition der *holding*-Abbildung ergibt sich, dass von *location-sets* aus, die als *holding* markiert sind, Eingabevariablen erreicht werden können. l_1 bis l_n in den Regeln stellen *location-sets* dar, k_1 bis k_n Konstanten, s_1 bis s_n Anweisungen. Die Bedingung $overlaps(u,v)$ ist erfüllt, wenn die *location-sets* u und v sich überlappende Mengen von Speicherstellen bezeichnen. Auch diese Regeln sind so dargestellt, dass die Unterhalb der Trennlinie genannten Eigenschaften gelten, wenn die oberhalb dieser Linie aufgelisteten Bedingungen erfüllt sind. Dabei bezeichnet s_n im Text die Anweisung, die in der Menge der Bedingungen an Stelle n steht, s_1 ist also die erste Zuweisung, s_2 die zweite.

Die folgenden Regeln decken den Fall ab, dass eine indirekte Zuweisung stattfindet und die Adresse der Zuweisung aus einer anderen Variablen bezogen wird.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}l_4 \wedge overlaps(l_1,l_3) \wedge holding(l_4)}{critical(*l_4{=}l_2)}$: Gibt es eine als kritisch betrachtete Anweisung $*l_1 = l_2$ und wird dem l_1 überlappenden *location-set* l_3 der Wert von l_4 zugewiesen, so können diese Zuweisungen zusammengefasst als $*l_4 = l_2$ dargestellt werden. Diese Anweisung wird als kritisch betrachtet, wenn durch Dereferenzieren von l_4 eine Eingabevariable erreicht werden kann.

- $\frac{critical(*l_1{=}l_2) \wedge s_2{=}(l_3{=}l_4{\pm}k_1 \vee l_3{=}k_1{\pm}l_4) \wedge overlaps(l_1,l_3) \wedge holding(l_4)}{summary(*l_1{=}l_2) \wedge critical(s_2)}$: Diese Anweisungen können nicht zusammengefasst werden. Daher wird die kritische Zuweisung in die *summary* aufgenommen und s_2 als kritisch markiert. Auch diese Regel wird nur angewendet, wenn von der Quellvariable von s_2 aus eine Eingabevariable erreicht werden kann.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}k_1 \wedge overlaps(l_1,l_3)}{summary(*l_1{=}l_2) \wedge critical(l_3{=}k_1)}$: Auch diese Kombination von Anweisungen kann nicht zusammengefasst werden; daher wird $*l_1 = l_2$ in die *summary* übernommen und $l_3 = k_1$ als kritisch markiert.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}l_4{\pm}l_5 \wedge overlaps(l_1,l_3) \wedge holding(l_4) \wedge holding(l_5)}{summary(*l_1{=}l_2) \wedge critical(l_3{=}l_4{\pm}l_5)}$: Eine Kombination dieser Zuweisungstypen kann nicht zusammengefasst werden. Daher wird die kritische Zuweisung in die *summary* aufgenommen und $l_3 = l_4 \pm l_5$ als kritisch markiert. Dies geschieht allerdings nur, wenn von $l_3 = l_4 \pm l_5$ aus Eingabevariablen erreicht werden können. Diese Bedigung muss für beide *location-sets* l_4 und l_5 erfüllt sein.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}*(\&l_4{+}offset{+}k{*}stride) \wedge overlaps(l_1,l_3) \wedge holding(l_4)}{critical(*l'_4{=}l_2)}$: Bei dieser Regel wird aus $l_4 =< block, o, s >$ das *location-set* $l'_4 =< block, offset + o, gcd($ $stride, s) >$ gebildet und statt der zweiten Anweisung $l_3 = l'_4$ betrachtet. Diese beiden Anweisungen können zusammengefasst werden. Ist die *holding*-Eigenschaft für l_4 bzw. l'_4 erfüllt, so wird die neu erzeugte Zuweisung als kritisch markiert.
- $\frac{critical(*l_1{=}l_2) \wedge *(\&l_3{+}offset{+}k{*}stride){=}l_4 \wedge overlaps(l_1,l'_3) \wedge holding(l_4)}{critical(*l_4{=}l_2)}$: Zuerst wird aus $l_3 =< block, o, s >$ das *location-set* $l'_3 =< block, offset + o, gcd(stride,$ $s) >$ gebildet. Überlappt dieses neu gebildete Set l_1 und ist die *holding*-Eigenschaft für l_4 erfüllt, so wird anschließend aus diesen beiden Zuweisungen die Anweisung $*l_4 = l_2$ erstellt und als kritisch markiert.
- $\frac{critical(*l_1{=}l_2) \wedge s_2{=}(l_3{=}*l_4 \vee l_3{=}*(l_4{+}offset{+}k{*}stride)) \wedge overlaps(l_1,l_3) \wedge holding(l_4)}{summary(*l_1{=}l_2) \wedge critical(s_2)}$: Zwei Dereferenzierungen lassen sich nicht in einer Zuweisung zusammenfassen. Kann über diese Dereferenzierungen jedoch eine Eingabevariable erreicht werden, so wird $*l_1 = l_2$ in die *summary* aufgenommen und die Analyse bei der Anweisung s_2 fortgesetzt. Um letzteres zu erreichen, wird s_2 als kritisch markiert.

Die gleichen Regeln werden auch angewendet, wenn es sich bei der betrachteten kritischen Zuweisung nicht um den Typ LCOMPLEX, sondern um eine LXCOMPLEX-Zuweisung handelt. In diesem Fall ist der Typ einer neu erzeugten Zuweisung ebenfalls LXCOMPLEX.

Es ist auch möglich, dass bei einem indirekten Speicherzugriff die Adresse direkt im dereferenzierten *location-set* gespeichert wird. Diese Kombination von Anweisungen wird durch die Regel $\frac{critical(*l_1{=}l_2) \wedge l_3{=}\&l_4 \wedge overlaps(l_1,l_3)}{critical(l_4{=}l_2)}$ abgedeckt. Für eine kritische Zuweisung vom Typ LXCOMPLEX wird die Regel abgewandelt zu $\frac{critical(*(l_1{+}offset{+}k{*}stride){=}l_2) \wedge l_3{=}\&l_4 \wedge overlaps(l_1,l_3)}{critical(*(\&l_4{+}offset{+}k{*}stride){=}l_2)}$. So wird diese Kombination von Zuweisungen durch eine einzige, direkte Zuweisung ausgedrückt.

Ist die Quellvariable einer indirekten Zuweisung die Zielvariable von einer anderen Zuweisung, so können die folgenden Regeln angewendet werden:

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}l_4 \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{critical(*l_1{=}l_4)}$: Bei dieser Kombination von Anweisungen kann der Zielvariablen von $*l_1$ direkt l_4 zugewiesen werden. Die neue Anweisung wird als kritisch markiert und so die Analyse an dieser Stelle fortgeführt, wenn von l_4 aus eine Eingabevariable erreicht werden kann.
- $\frac{critical(*l_1{=}l_2) \wedge s_2{=}(l_3{=}l_4{\pm}k_1 \vee l_3{=}k_1{\pm}l_4) \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{summary(*l_1{=}l_2) \wedge critical(s_2)}$: Anweisungen dieser Typen können nicht zusammengefasst werden. Daher wird die kritische Zuweisung in die *summary* aufgenommen und s_2 als kritisch markiert. Bedingung ist, dass von der Quellvariable der Anweisung s_2 aus eine Eingabevariable erreicht werden kann.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}k_1 \wedge overlaps(l_2,l_3)}{summary(*l_1{=}l_2) \wedge critical(l_3{=}k_1)}$: Auch diese Kombination von Anweisungen kann nicht zusammengefasst werden; daher wird $*l_1 = l_2$ in die *summary* übernommen und $l_3 = k_1$ als kritisch markiert.

- $\frac{critical(*l_1{=}l_2) \wedge s_2{=}(l_3{=}l_4 \pm l_5) \wedge overlaps(l_2,l_3) \wedge holding(l_4) \wedge holding(l_5)}{summary(*l_1{=}l_2) \wedge critical(l_3{=}l_4 \pm l_5)}$: Diese Anweisungstypen können nicht kombiniert werden. Daher wird die kritische Zuweisung in die *summary* aufgenommen und s_2 als kritisch markiert. Dies geschieht allerdings nur, wenn von $l_3 = l_4 \pm l_5$ aus Eingabevariablen erreicht werden können. Diese Bedigung muss für beide *location-sets* l_4 und l_5 erfüllt sein.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}*(\&l_4{+}offset{+}k{*}stride) \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{critical(*l_1{=}l_4')}$: Für Kombination von Anweisungen dieser Typen wird aus $l_4 =< block, o, s >$ zunächst das *location-set* $l_4' =< block, offset + o, gcd(stride, s) >$ gebildet. Ist es möglich, durch Dereferenzieren dieses Sets externe Variablen zu erreichen, so werden beide Anweisungen zusammengefasst und die neu entstandene Zuweisung $*l_1 = l_4'$ als kritisch markiert.

- $\frac{critical(*l_1{=}l_2) \wedge *(\&l_3{+}offset{+}k{*}stride){=}l_4 \wedge overlaps(l_2,l_3') \wedge holding(l_4)}{critical(*l_1{=}l_4)}$: Das *location-set* l_3' bilet sich aus $l_3 =< block, o, s >$ zu $l_3' =< block, offset + o, gcd(stride, s) >$. Überlappt dieses Set l_2, so können beide Zuweisungen zu $*l_1 = l_4$ zusammengefasst werden. Diese neue Zuweisung wird als kritisch markiert, wenn von l_4 aus auf nicht-lokale Variablen zugegriffen werden kann. Somit wird die Analyse mit $*l_1 = l_4$ fortgesetzt.

- $\frac{critical(*l_1{=}l_2) \wedge l_3{=}\&l_4 \wedge overlaps(l_2,l_3)}{summary(*l_1{=}l_2) \wedge critical(l_3{=}\&l_4)}$: Eine solche Kombination von Zuweisungen kann nicht zusammengefasst werden; daher wird die Anweisung $*l_1 = l_2$ Teil der *summary* und $l_3 = \&l_4$ als kritisch markiert. So wird die Analyse bei letzterer Anweisung fortgeführt.

- $\frac{critical(*l_1{=}l_2) \wedge s_2{=}(l_3{=}*l_4 \vee l_3{=}*(l_4{+}offset{+}k{*}stride)) \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{summary(*l_1{=}l_2) \wedge critical(s_2)}$: Da zwei Dereferenzierungen nicht kombiniert werden können, wird $*l_1 = l_2$ in die *summary* übernommen und die Analyse mit der Anweisung s_2 fortgesetzt, sofern es möglich ist, dass durch l_4 eine Eingabevariable erreicht werden kann.

Diese Regeln gelten auch, wenn der Zuweisungstyp der betrachteten kritischen Anweisung LXCOMPLEX ist. In diesem Fall hat auch eine neu erstellte Zuweisung den Typ LXCOMPLEX.

Es ist ebenfalls möglich, dass sowohl Quell- als auch Zielvariablen einer Anweisung Eingabevariablen darstellen. Eine solche Zuweisung wird direkt in die *summary* übernommen. Ausgedrückt wird dies durch die Regeln

- $\frac{critical(*l_1{=}l_2) \wedge input(l_1) \wedge input(l_2)}{summary(*l_1{=}l_2)}$

- $\frac{critical(*(l_1+offset+k*stride)=l_2) \wedge input(l_1) \wedge input(l_2)}{summary(*(l_1+offset+k*stride)=l_2)}$

Diese Regeln sind explizit nur für Zuweisungen vom Typ LCOMPLEX bzw. LXCOMPLEX aufgeführt; andere Zuweisungen werden in die *summary* übernommen, wenn nur die Quellvariablen Eingabevariablen darstellen; diese Fälle werden allerdings durch andere Regeln abgedeckt.

Es ist ebenfalls möglich, dass Anweisungen vom Typ $u = \&v$ als kritisch betrachtet werden. Da diese Zuweisungen nicht mit anderen Zuweisungen kombiniert werden können, werden sie direkt in die *summary* übernommen. Dies geschieht durch die Regel $\frac{critical(l_1=\&l_2)}{summary(l_1=\&l_2)}$. Ist zusätzlich die Eigenschaft $holding(l_2)$ erfüllt, so werden alle Anweisungen, die l_2 beschreiben, als kritisch markiert.

Für kritische Zuweisungen der Typen RCOMPLEX und RXCOMPLEX gelten folgende Regeln:

- $\frac{critical(l_1=*l_2) \wedge l_3=l_4 \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{critical(l_1=*l_4)}$: Die beiden genannten Zuweisungen lassen sich kombinieren zu $l_1 = *l_4$. Kann über l_4 auf eine externe Variable zugegriffen werden, so wird die neu erstellte Zuweisung als kritisch markiert und so in die Analyse einbezogen.
- $\frac{critical(l_1=*l_2) \wedge *(\&l_3+offset+k*stride)=l_4 \wedge overlaps(l_2,l_3') \wedge holding(l_4)}{critical(l_1=*l_4)}$: Zuerst wird aus $l_3 =< block, o, s >$ das *location-set* l_3' erstellt mit $l_3' =< block, offset + o, gcd(s, stride) >$ gebildet. Überlappt l_3' das Set l_2, so kann der Datenfluss zu $l_1 = *l_4$ kombiniert werden. Kann durch Dereferenzieren von l_4 auf eine externe Variable zugegriffen werden, so wird die neu erstellte Zuweisung anschließend als kritisch markiert und so in die Analyse einbezogen.
- $\frac{critical(l_1=*l_2) \wedge l_3=*l_4 \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{summary(l_1=*l_2) \wedge critical(l_3=*l_4)}$: Diese Kombination von Anweisungen kann nicht in nur eine Zuweisung umgewandelt werden. Sind diese Zuweisungen dennoch für die *summary* von Bedeutung, so wird $l_1 = *l_2$ in die *summary* eingefügt und die Analyse mit der Anweisung $l_3 = *l_4$ fortgesetzt. Diese Regel wird auch angewendet, wenn es sich statt $l_3 = *l_4$ um eine Zuweisung vom Typ RXCOMPLEX, PLUSCONST, MINUSCONST, CONSTPLUS oder CONSTMINUS handelt. Für Zuweisungen vom Typ CONSTANT wird die Abfrage $holding(l_4)$ weggelassen. AR_PLUS- und AR_MINUS-Zuweisungen führen nur zu Effekten auf die *summary* und die Menge der kritischen Anweisungen, wenn beide Quellvariablen als *holding* eingestuft sind.
- $\frac{critical(l_1=*l_2) \wedge input(l_2)}{summary(l_1=*l_2)}$: Wenn eine Eingabevariable erreicht wird, so wird die kritische Zuweisung in die *summary* aufgenommen.
- $\frac{critical(l_1=*l_2) \wedge l_3=\&l_4 \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{critical(l_1=l_4)}$: In diesem Fall ist die Zielvariable l_4 von l_2 bekannt, daher können beide Anweisungen zu $l_1 = l_4$ zusammengefasst werden. Kann über l_4 auf eine externe Variable zugegriffen

werden, so wird die neu erzeugte Zuweisung als kritisch markiert. Daraufhin wird die Analyse an dieser Zuweisung fortgesetzt.

Die Regeln sind jeweils für RCOMPLEX-Zuweisungen dargestellt; Zuweisungen vom Typ RXCOMPLEX werden jedoch identisch behandelt; neu erzeugte Zuweisungen sind dann allerdings ebenfalls vom Typ RXCOMPLEX beziehungsweise im Fall der letzten Regel RXSIMPLE.

Für Anweisungen vom Typ SIMPLE, XSIMPLE, RXSIMPLE, PLUSCONST, MINUSCONST, CONSTPLUS, CONSTMINUS, CONSTANT, AR_PLUS und AR_MINUS gibt es die folgenden Regeln:

- Ist die Variable auf der rechten Seite der Zuweisung eine Eingabevariable, so wird die Zuweisung in die *summary* übernommen. Bei Anweisungen vom Typ AR_PLUS und AR_MINUS geschieht dies nur, wenn beide Variablen Eingabevariablen sind. Eine kritische Anweisung vom Typ CONSTANT wird immer zur *summary* hinzugefügt.

- Anweisungen vom Typ XSIMPLE und RXSIMPLE werden in eine SIMPLE-Zuweisung umgewandelt. Dazu wird das sich durch die Parameter $offset$ und $stride$ effektiv ergebende Quell- bzw. Ziel-*location-set* berechnet.

- $\frac{critical(l_1{=}l_2) \wedge l_3{=}\&l_4 \wedge overlaps(l_2,l_3)}{critical(l_1{=}\&l_4)}$: Die beiden Anweisungen lassen sich zusammenfassen. Eine solche Anweisung wird immer als kritisch betrachtet, wenn bereits $l_1 = l_2$ kritisch ist. Diese Regel wird auch für andere Zuweisungstypen als SIMPLE für $l_1 = l_2$ angewendet, wenn diese Anweisungen keine Dereferenzierung und keinen Adressoperator enthalten. In diesen Fällen allerdings kann der Datenfluss nicht zusammengefasst werden. Daher wird die bisherige kritische Anweisung in die *summary* kopiert und die Zuweisung $l_3 = \&l_4$ als kritisch markiert. Bei kritischen Anweisungen vom Typ AR_PLUS und AR_MINUS wird geprüft, ob l_3 mindestens eine der Quellvariablen der bisherigen kritischen Anweisung überlappt.

- $\frac{critical(l_1{=}l_2) \wedge l_3{=}{*}l_4 \wedge overlaps(l_2,l_3) \wedge holding(l_4)}{critical(l_1{=}{*}l_4)}$: Die beiden Anweisungen lassen sich zusammenfassen. Das gilt auch, wenn statt $l_3 = *l_4$ eine Zuweisung $l_3 = *(l_4 + offset + k*stride)$ betrachtet wird. In diesem Fall kann sich eine Zuweisung $l_1 = *(l_4 + offset + k*stride)$ ergeben. Eine solche erzeugte Anweisung wird als kritisch betrachtet, wenn bereits $l_1 = l_2$ kritisch ist und sich von l_4 aus auf eine nicht-lokale Variable zugreifen lässt. Diese Regel wird auch für andere Zuweisungstypen als SIMPLE für $l_1 = l_2$ angewendet, sofern diese weder einen Dereferenzierungs- noch einen Adressoperator beinhalten. Da in diesen Fällen allerdings der Datenfluss nicht zusammengefasst werden kann, wird die bisherige kritische Anweisung in die *summary* kopiert und die Zuweisung $l_3 = *l_4$ als kritisch markiert. Bei kritischen Anweisungen vom Typ AR_PLUS und AR_MINUS wird geprüft, ob l_3 mindestens eine der Quellvariablen der kritischen Anweisung überlappt.

- Eine Kombination s_1, s_2 von Anweisungen, die beide weder Adress- noch Dereferenzierungsoperator enthalten kann zusammengefasst werden, wenn es sich bei einer der Anweisungen um eine Zuweisung vom Typ SIMPLE handelt und sich die Quellvariable von s_1 mit der Zielvariablen von s_2 überlappt. Ist s_1 als kritisch markiert und kann durch die Quellvariable von s_2 eine Eingabevariable erreicht werden, so wird die neu erzeugte Anweisung ebenfalls als kritisch betrachtet. Kann ein solcher Datenfluss nicht zusammengefasst werden, so wird s_1 in die *summary* aufgenommen und s_2 wird Teil der Menge kritischer Anweisungen.

- Eine Anweisung vom Typ CONSTANT kann mit Anweisungen vom Typ SIMPLE, AR_PLUS und AR_MINUS kombiniert werden; so kann eine neue Anweisung vom Typ PLUSCONST, MINUSCONST etc. erzeugt werden. Diese Anweisung wird als kritisch betrachtet, wenn die betreffende Anweisung vom Typ AR_PLUS bzw. AR_MINUS bereits als kritisch markiert war und es einen Pfad im Kontrollflussgraph der Funktion von der CONSTANT-Anweisung aus zur kritischen Anweisung gibt.

Es lässt sich erkennen, dass diese Regeln alle Kombinationen von jeweils zwei Anweisungen abdecken, bei der ein Datenfluss von der einen zur anderen Anweisung auftreten kann. Damit kann durch wiederholtes Anwenden dieser Regeln der gesamte Datenfluss in einer Funktion analysiert werden. Dies ist möglich, da die neu erzeugte Anweisungen, die eine Kombination mehrerer Anweisungen darstellen, ebenfalls als kritisch markiert werden; so werden sie im nächsten Iterationsschritt analysiert. Diese Iteration bricht erst ab, wenn während einem ganzen Durchlauf durch die Anweisungen einer Funktion keine weiteren Anweisungen mehr in die *summary* aufgenommen werden und auch keine zusätzlichen Zuweisungen mehr als kritisch markiert werden.

Vereinfachen der summary

Die zu diesem Zeitpunkt berechnete *summary* besitzt weiteres Optimierungspotenzial. Ähnlich wie in der Phase der Codeumwandlung kann es vorkommen, dass es mehrere sich überlappende *location-sets* gibt. Diese lassen sich auf gleiche Weise vereinigen. Das wiederum kann zu Anweisungen führen, die selbst keinen Effekt haben oder die redundant sind. Erstere Art kann vollständig, letztere Anweisungen können bis auf eine Kopie entfernt werden. Da innerhalb der erstellten *summary* kein Kontrollfluss mehr betrachtet wird, ist dieser für die Beurteilung der Redundanz ebenfalls bedeutungslos; zwei identische Anweisungen sind auch dann redundant, wenn es zwischen diesen beiden Anweisungen keinen Kontrollflusspfad gibt.
Weiterhin kann es vorkommen, dass Zweige im Datenfluss mit in die *summary* aufgenommen worden sind, von denen aus keine Eingabevariable zu erreichen ist. Das liegt vor allem an der konservativen Abschätzung der *holding*-Eigenschaft bei

der Suche nach “kritischen” Anweisungen; in dieser Phase kann ein *location-set* als *holding* eingestuft werden, obwohl selbst in beliebiger Kombination mit anderen im Funktionskörper enthaltenen Anweisungen nicht auf eine externe Variable zugegriffen werden kann. Diese Zweige sind folglich überflüssig und können entfernt werden. Das geschieht durch einen iterativen Algorithmus, der den möglichen Datenfluss in der *summary* betrachtet, allerdings ohne Informationen zum Kontrollfluss. In jedem Iterationsschritt werden alle *location-sets* markiert, die sich auf eine Eingabevariable beziehen. Ferner werden alle *location-sets* markiert, die Ziel einer Zuweisung sind. Ebenso werden alle Sets markiert, die sich auf Variablen beziehen, die mit dem Adressoperator angesprochen werden. Unmarkiert bleiben also nur die *location-sets*, die sich nicht auf eine Eingabevariable beziehen und weder direkt noch indirekt beschrieben werden können. Anschließend werden alle Anweisungen aus der *summary* entfernt, die mindestens eine unbekannte Quellvariable haben.
Da durch das Entfernen von einzelnen Anweisungen nun weitere Anweisungen überflüssig werden können, wird dieser Iterationsschritt wiederholt, bis keine weiteren Anweisungen mehr aus der *summary* entfernt werden.

3.3.6 Einfügen von Summaries

Durch das Einfügen von *summaries* an Stelle von Funktionsaufrufen werden die Anweisungen, die Effekte auf eine aufgerufene Funktion hervorrufen, direkt in die aufrufende Funktion übertragen. Dadurch sind die Effekte der aufgerufenen Funktion der aufrufenden Funktion vollständig bekannt. Beim Einfügen einer solchen *summary* wird zunächst eine Kopie dieser *summary* erstellt. Dabei werden alle lokalen Variablen aus der *summary* der aufgerufenen Funktion durch neue, temporäre Variablen ersetzt. Nur globale Variablen behalten den ursprünglichen Namen. Grund für die Beibehaltung globaler Variablen ist, dass diese auch über die aufrufende Funktion hinaus sichtbar sind. Daher müssen diese Zuweisungen zu globalen Variablen auch in der *summary* der aufrufenden Funktion vorhanden sein, bei einer Ersetzung durch temporäre Variablen wäre dies nicht mehr der Fall.
Durch die Nutzung von temporären Variablen werden die Auswirkungen für jeden Funktionsaufruf separat behandelt. Anschließend werden Anweisungen eingefügt, die die Argumente, mit denen die Funktion aufgerufen wurde, in die jeweiligen temporären Variablen der eingefügten *summary* kopieren. Bei diesen temporären Variablen handelt es sich um die Eingabevariablen der aufgerufenen Funktion; dabei handelt es sich neben globalen Variablen ausschließlich um den in Kapitel 3.3.2 eingeführten Stack für Funktionsargumente. Handelt es sich bei der *summary* um die Effekte von einem rekursiven Zyklus, so wird die Zuweisung zu dem Speicherbereich durchgeführt, der den Stack derjenigen Funktion repräsentiert, durch die in den Zyklus eingetreten wird.

Für die aufgerufene Funktion dagegen werden alle Aufrufkontexte verschmolzen, so dass für die lokalen Variablen dieser Funktion ausreichend konservative *Points-*

to-Werte berechnet werden. Dies geschieht dadurch, dass die Argumente auch den ursprünglichen Variablen der aufgerufenen Funktion zugewiesen werden. Weiterhin wird eine Anweisung eingefügt, die den Rückgabewert der Funktion - der in einem definierten Speicherbereich abgelegt wird - dem Ziel-*location-set* der simpleStmt-Anweisung mit dem Funktionsaufruf zuweist. In dem in Listing 3.6 dargestellten Beispiel wird in die Funktion g zweimal eine Summary der Funktion f eingefügt, die aus der Anweisung *a=b besteht. Die Funktionskörper bestehen aus simpleStmt-Objekten, die Zuweisung der Rückgabewerte ist nicht dargestellt, da diese nicht genutzt werden. Der ursprüngliche Funktionsaufruf wird anschließend nicht mehr betrachtet.

```
void f(int *a, int b) {
  *a=b;
}

/* originale Funktion g */
void g() {
  int i, j, k, l;
  f(&i, j);

  f(&k, l);
}

/* Funktion g mit summaries statt Funktionsaufrufen */
void g() {
  int i, j, k, l;
  a1=&i;
  b1=j;
  *a1=b1;
  a=&i;
  b=j;

  a2=&k;
  b2=l;
  *a2=b2;
  a=&k;
  b=k;
}
```

Listing 3.6: Einfügen von Summaries

Die Anweisungen der *summary* enthalten weiterhin die während der Codeumwandlungsphase (Kapitel 3.3.2) erstellten Bindungen zu einem IR_Stmt. Da sich diese Bindung auf den Kontrollfluss innerhalb der aufgerufenen Unterfunktion bezieht, ist diese Bindung für die in die aufrufende Funktion eingefügten Zuweisungen natürlich ungültig. Daher wird diese Bindung so modifiziert, dass die eingefügten Anweisungen zu dem IR_Stmt gebunden werden, das den Funktionsaufruf enthält. Damit stehen die Anweisungen aus der *summary* an gleicher Stelle in der aufrufenden Funktion wie der Funktionsaufruf, den sie ersetzen.

Für einen indirekten Funktionsaufruf werden die *summaries* von allen möglichen Zielfunktionen eingefügt; die Menge der Zielfunktionen ergibt sich aus der *Points-to*-Menge des im Aufruf verwendeten Funktionszeigers.

3.3.7 Erzeugen der Points-to Beziehungen

Dieser Schritt wird in der in Kapitel 3.3.4 erwähnten *Top-down*-Phase ausgeführt. In diesem Schritt werden die eigentlichen Aliasbeziehungen berechnet, nach dem das zu analysierende Programm durch das Einfügen von *summaries* so modifiziert wurde, dass interprozedurale Effekte durch eine Funktion nicht mehr berücksichtigt werden müssen. Daher werden in dieser Phase der Analyse die *Points-to*-Beziehungen für jede Funktion einzeln berechnet. Dazu werden alle Anweisungen im Funktionskörper durchlaufen und deren Auswirkungen auf die Aliasbeziehungen im *Points-to*-Graph eingetragen. Dies geschieht durch den Aufruf der Methode localPA der in Abbildung 3.10 abgebildeten Klasse IR_Function. Diese Methode fügt die im umgewandelten Code vorhandenen Zuweisungen in den *Points-to*-Graph ein; dieser wird durch die Klasse PTG (Abbildung 3.11) dargestellt. Die in dieser Klasse enthaltene Methode PTG::addAssignment fügt die durch eine einzelne Zuweisung entstehenden Aliasse zu den *Points-to*-Beziehungen hinzu.

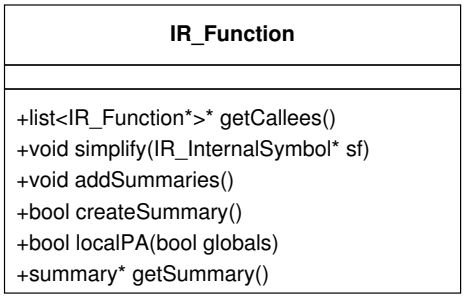


Abbildung 3.10: Klasse IR_Function

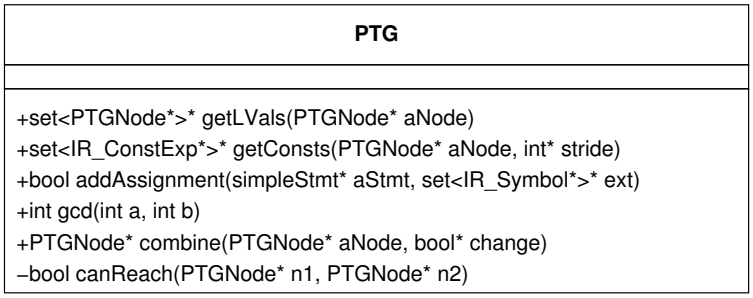


Abbildung 3.11: Klasse PTG

Dabei werden in dieser Phase nur die lokalen Effekte von Anweisungen betrachtet; die Auswirkungen auf eine aufrufende Funktion sind bereits durch das Einfügen der *summary* an der entsprechenden Aufrufstelle Analysiert. Lediglich die Hauptfunktion wird vollständig analysiert; für diese Funktion wurde ja keine *summary* erstellt, da es nur einen einzigen, bekannten Aufrufkontext gibt. Die folgenden Arten von Anweisungen können einen Effekt auf die aufrufende Funktion haben:

1. **Zuweisung zu globalen Variablen:**
   Diese Zuweisungen sind für die aufrufende Funktion sichtbar und entsprechend Teil der *summary*.

2. **Indirekte Zuweisung:**
   Eine Anweisung der Form $*u = v$ kann einen Effekt auf die aufrufende Funktion haben, wenn u auf eine nicht lokal definierte Variable zeigen kann.

Würden die gezeigten Zuweisungen vollständig betrachtet, so hätte dies den Effekt, dass alle Aufrufkontexte verschmolzen würden. Das liegt daran, dass für die lokale Analyse einer Funktion alle Aufrufkontexte verschmolzen werden, um ausreichend konservative *Points-to*-Werte zu berechnen. Würden mit diesen verschmolzenen Aufrufkontexten nun Effekte auf aufrufende Funktionen berechnet werden, so würden diese Effekte die Vereinigung der Effekte aller Aufrufkontexte sein; die Analyse würde also kontextinsensitiv arbeiten.

Die Auswirkungen durch einzelne Zuweisungen lassen sich durch feste Regeln ausdrücken. Die folgenden Funktionen werden von diesen Regeln verwendet:

- $LVals(x)$ bezeichnet alle bekannten *Points-to*-Werte von *location-set* x. Rückgabewert ist eine Liste mit allen Einträgen der Liste baseElements von x sowie allen x zugewiesenen *location-sets*. Letztere Sets lassen sich im *Points-to*-Graph durch das Folgen der pointsTo-Kanten der enthaltenen PTGNode-Instanzen erreichen.

- $getConsts(x)$ liefert eine Liste mit allen möglichen Konstanten, die das *location-set* x enthalten kann, zurück. Diese Methode ähnelt in der Implementierung *LVals*, jedoch werden statt der Basiselemente der Knoten die dort gespeicherten Konstanten zurückgeliefert.

- $getConstStride(x)$ liefert 0 zurück, wenn das *location-set* x nur bekannte Ganzzahlwerte enthält. Andernfalls wird ein Wert s zurückgeliefert, so dass die Werte, die x annehmen kann, in der Menge

$$\bigcup_{c \in getConsts(x), z \in Z} c + zs$$

  enthalten sind. Wie bei *LVals* und *getConsts* werden auch die Werte von anderen *location-sets* berücksichtigt, wenn diese x zugewiesen wurden.

- $gcd(x,y)$ liefert den größten gemeinsamen Teiler von x und y zurück.
- $canReach(x,y)$ liefert $true$ zurück wenn sich die *location-sets* x und y überlappen oder wenn es eine Zuweisungskette $x = t_1$, $t_1 = t_2$, ..., $t_n = y$ gibt. Ansonsten wird der Wert $false$ zurückgeliefert.

Weiterhin wird vor dem Einfügen eines neuen *location-sets* in den *Points-to*-Graph geprüft, ob es überlappende Sets gibt. Ist dies der Fall, werden diese Sets inklusive dem neuen Set vereinigt. Dies geschieht durch die Methode PTG::combine. Diese Methode wird für jedes im PTG::addAssignment übergebenen simpleStmt enthaltene *location-set* aufgerufen.
Die nun aufgeführten Regeln beziehen sich jeweils auf einen bestimmten Zuweisungstyp bzw. eine Gruppe von Zuweisungstypen der Klasse simpleStmt. Anweisungen vom Typ CALLSTMT werden nicht betrachtet, sind diese Anweisungen doch durch jeweils eine *summary* ersetzt.

- BASE
  Bei einer Zuweisung $u = \&v$ wird v als *Points-to*-Wert von u eingetragen, d.h. in die Liste baseElements von u aufgenommen.
- SIMPLE
  Bei Zuweisungen $u = v$ wird die Liste pointsTo von u um v erweitert. Durch den Typ der Zuweisung bzw. der Anzahl der zugewiesenen Bytes kann es vorkommen, dass auch ein *location-set* u' beschrieben wird, das sich auf den gleichen Speicherbereich bezieht wie u, jedoch einen höheren Offset hat. Um diese Zuweisungen abzudecken, wird der Wert $conststride$ für u' auf den Wert eins gesetzt; das bedeutet, dass u' jeden beliebigen Ganzzahlwert enthalten kann. Mögliche Zuweisungen von Zeigerwerten zu u' werden nicht berücksichtigt, da in diesen Fällen - Typumwandlung und Zuweisungen von struct-Variablen - während der Codeumwandlung explizite Zuweisungen eingefügt werden.
- XSIMPLE
  Für Zuweisungen der Form $*(\&u + offset + k * stride) = v$ mit $u =< block, o, s >$ wird zunächst das *location-set* $u' =< block, offset + o, gcd(stride, s) >$ gebildet. Diesem Set u' wird dann wie bei einer Zuweisung vom Typ SIMPLE das Set v zugewiesen.
- RXSIMPLE
  Für diese Zuweisungen $u = *(\&v + offset + k * stride)$, $v =< block, o, s >$ wird v' gebildet mit $v' =< block, offset + o, gcd(stride, s) >$. Anschließend wird die Anweisung $u = v'$ betrachtet, d.h., die Regel für SIMPLE angewendet.
- LCOMPLEX
  Bei Zuweisungen $*u = v$ wird zunächst die Menge $L = LVals(u)$ bestimmt.

Wird eine andere Funktion als die Hauptfunktion analysiert, werden aus L alle Elemente entfernt, die Speicherstellen von nicht lokalen Variablen darstellen. Anschließend wird für alle $l \in L$ die Zuweisung $l = v$ entsprechend der Regel für SIMPLE ausgeführt.

- LXCOMPLEX
  Für diese Zuweisungen $*(u + offset + k * stride) = v$ wird ebenfalls zunächst die Menge $L = LVals(u)$ bestimmt und gegebenenfalls nicht-lokale Elemente aus L entfernt. Anschließend wird für alle $l =< block, o, s >$ aus L das *location-set* das Set $l' =< block, offset + o, gcd(stride, s) >$ gebildet und die Zuweisung $l' = v$ ausgeführt.

- RCOMPLEX
  Für Zuweisungen $u = *v$ wird, sofern nicht bereits vorhanden, ein Knoten für $*v$ in den *Points-to*-Graph eingefügt. Dieser Knoten wird in die Liste pointsTo von u aufgenommen. Anschließend wird die Menge $L = LVals(v)$ bestimmt und alle l aus L in die Liste pointsTo des Knotens für $*v$ aufgenommen.

- RXCOMPLEX
  Für Zuweisungen $u = *(v + offset + k * stride)$ wird anders als bei RCOMPLEX kein Knoten für $*v$ erzeugt. Stattdessen wird direkt die Menge $L = LVals(v)$ bestimmt und für jedes $l =< block, o, s >$ aus L das *location-set* $l' =< block, offset + o, gcd(stride, s) >$ berechnet. Anschließend wird für jedes berechnete l' die Zuweisung $u = l'$ gemäß der Regel von SIMPLE ausgeführt.

- CONSTANT
  Zuweisungen der Form $x = k$ werden betrachtet, wenn k ein Ganzzahlwert ist. Dann wird k in die Menge von $getConsts(x)$ aufgenommen. Zusätzlich wird angenommen, dass k die absolute Adresse einer Variablen darstellt. Da durch absolute Adressierung auf jede Stelle im Speicher zugegriffen werden kann, diese Zugriffe aber nicht auf deklarierte Speicherbereiche abgebildet werden können, wird ein Speicherbereich a definiert, der den gesamten Adressraum überdeckt und auf den Zugriffe über absolute Adressen stattfinden. Überlappungen zu anderen Speicherbereichen werden nicht angenommen. Daher wird bei jeder Integerzuweisung zusätzlich die Anweisung $x = \& < a, k, 0 >$ gebildet und nach der Regel für BASE analysiert.
  Handelt es sich nicht um eine Integer-Zuweisung, so wird angenommen, dass x jeden beliebigen Wert annehmen kann, d.h. $getConstStride(x)$ wird den Wert 1 zurückliefern. Ebenso wird angenommen, dass x auf jede Stelle des als a definierten Speicherbereichs zeigen kann; es wird also zusätzlich die Anweisung $x = \& < a, 0, 1 >$ betrachtet.

- AR_PLUS, AR_MINUS
  Für Zuweisungen $u = v + w$ beziehungsweise $u = v - w$ wird zunächst festgestellt, wie viele Bytes zugewiesen werden. Ist diese Zahl größer als eine

Zeigervariable, so werden die ebenfalls beschriebenen Speicherstellen u' aus dem Speicherbereich, auf den sich das *location-set u* bezieht, als unbekannt angenommen. Dies geschieht durch das Setzen des Wertes *constStride* von u' auf 1, d.h., in Zukunft gilt $getConstStride(u') = 1$. Anschließend wird die eigentliche Zuweisung betrachtet. Dabei kann es sich sowohl bei v als auch bei w um Ganzzahlwerte als auch um Zeiger handeln. Eine Addition von zwei Zeigerwerten ist in der Programmiersprache *C* jedoch nicht möglich, folglich kann dieser Fall auch im umgewandelten Code nicht auftreten. Es wird also unterschieden zwischen einer Addition von zwei Konstanten sowie der Addition eines Offsets und eines Zeigers. Da nicht klar ist, ob ein *location-set* einen Zeiger oder eine Konstante repräsentiert, werden jeweils beide Fälle betrachtet. Daraus ergeben sich die Kombinationen:

1. Addition bzw. Subtraktion der Konstanten v und w. Bestimmt werden die Mengen $C_v = getConsts(v)$ und $C_w = getConsts(w)$ sowie die Werte $s_v = getConstStride(v)$ und $s_w = getConstStride(w)$. Der Wert für $constStride(u)$ berechnet sich zu $gcd(getConstStride(u), gcd(s_v, s_w))$. Damit wird ein eventuell bisher vorhandener Wert von *constStride* einbezogen. Anschließend wird die Menge $S = \{s_1 + s_2 | s_1 \in C_v, s_2 \in C_w\}$ bzw. $S = \{s_1 - s_2 | s_1 \in C_v, s_2 \in C_w\}$ gebildet und der Menge der Konstanten von u hinzugefügt. Liefert $canReach(v,u)$ oder $canReach(w,u)$ den Wert *true* zurück, so gibt es einen zyklischen Datenfluss und es kann auf die Zielvariable wiederholt eine Konstante addiert werden. In diesen Fällen wird der Parameter *constStride* von u auf den Wert von $gcd(conststride(u), r)$ gesetzt; dabei ist r der größte gemeinsame Teiler von allen Elementen der Menge S.
2. Zum bzw. vom Zeiger v wird Konstante w addiert bzw. subtrahiert. Bestimmt werden die Mengen $L = LVals(v)$, $C = getConsts(w)$ sowie $s = getConstStride(w)$. Anschließend wird für alle $l =< block, offset, stride >\in L$ und $c \in C$ das *location-set* l' gebildet mit $l' =< block, offset + c, gcd(stride, s) >$ bzw. $l' =< block, offset - c, gcd(stride, s) >$. Auch in diesem Fall wird mittels der Funktion *canReach* festgestellt, ob die Addition eines Offsets im Programm mehrfach ausgeführt werden kann; in diesem Fall gilt $l' =< block, offset + c, gcd(stride, gcd(s, c))$ bzw. $l' =< block, offset - c, gcd(stride, gcd(s, c))$. Dieses neu gebildete Element wird in die Liste baseElements von u aufgenommen. Der Fall, dass es sich bei w um einen Zeiger handelt, zu dem die Konstante v addiert wird, wird analog behandelt.

- PLUSCONST, MINUSCONST, CONSTPLUS, CONSTMINUS
  Diese Zuweisungen bestehen aus einem Ziel-Set, einem Quell-Set und einer Konstante. Ein solche Zuweisung wird wie in der Regel für AR_PLUS- bzw. AR_MINUS bearbeitet, die Werte des einen Operanden ergeben sich aus der Konstanten.

Bis auf Anweisungen vom Typ LCOMPLEX und LXCOMPLEX wird in anderen Funktionen als main für ein simpleStmt PTG::addAssignment nur dann aufgerufen, wenn zu der linken Seite der Zuweisung ein lokal definierter Speicherbereich gehört; dazu gehören auch die beim Einfügen einer *summary* definierten Variablen.
Nach Möglichkeit werden Zuweisungen nur einmal analysiert, doch dies ist bei den Anweisungstypen, deren Regeln einen Aufruf von *LVals* besitzen, nicht möglich. Hier kann die Ausführung der Regel für eine Zuweisung das Ergebnis einer anderen Anweisung verändern. Daher sind diese Zuweisungen wiederholt auszuführen bis ein Fixpunkt erreicht ist. Ein solcher Fixpunkt wird erreicht, da die Anzahl der *location-sets* aufgrund der Verwendung von Schrittweiten begrenzt ist und die Menge der *Points-to*-Beziehungen monoton wächst; letzteres liegt darin begründet, dass keine *strong updates* (Kapitel 2.2.4) durchgeführt werden.

3.3.8 Heap-Objekte

Bei Heap-Blöcken handelt es sich um Speicherbereiche beliebiger Größe, die dynamisch während der Laufzeit eines Programms alloziert werden können. Dazu stehen in *C* spezielle Bibliotheksfunktionen wie z.B. malloc bereit. Die Zahl der Heap-Blöcke kann theoretisch unendlich groß sein. Es ist daher unumgänglich, zusätzliche Aliasse zwischen verschiedenen Heap-Blöcken anzunehmen. Dabei ist ein guter Kompromiss zu finden zwischen induzierter Ungenauigkeit der Analyseergebnisse und der Geschwindigkeit der Analyse [LA03] [NKH04b]. Untersuchungen von Programmen haben ergeben, dass Heap-Speicher häufig in Zusammenhang mit rekursiven (dynamischen) Datenstrukturen verwendet wird. Dazu zählt z.B. die Datenstruktur einer Liste, die aus einzelnen Elementen besteht, die jeweils einen Datenwert sowie einen Zeiger auf ein folgendes Listenelement enthalten. Häufig arbeiten Funktionen iterativ oder rekursiv auf solchen Datenstrukturen; als Beispiel sei eine Funktion für die genannte Listenstruktur erwähnt, die ermittelt, ob sich ein bestimmter Wert in der Liste befindet. Diese Suche wird vom Listenkopf aus alle Elemente der Liste durchsuchen, bis ein bestimmter Datenwert gefunden wird. Ist das Listenende erreicht, bricht die Suche ab. Dies geschieht etwa durch eine Schleife, in der einem - nur lokal kopierten - Zeiger auf ein Listenelement jeweils die Adresse des Nachfolgeelements zugewiesen wird (Siehe Listing 3.7).
In diesem Beispiel ist zu sehen, dass start praktisch auf jedes Listenelement zeigen kann, wenn die Annahme getroffen wird, dass das der Funktion übergebene Element den Listenkopf darstellt. Daher wird als Vereinfachung angenommen, dass nur vollständig getrennte Datenstrukturen unterschieden werden, nicht aber z.B. einzelne Elemente einer Liste. Dabei wird in realen Programmen eine Datenstruktur oft mit einer bestimmten Menge von Funktionen bearbeitet. Eine Annahme wäre daher, Heapblöcke nach der Funktion, in der sie erzeugt werden, zu unterscheiden. Diese Idee greift allerdings zu kurz, da häufig die Funktion malloc nicht direkt von den Funktionen, die eine Datenstruktur bearbeiten, aufgerufen wird, sondern erst in einer weiteren Unterfunktion. Diese Funktionen können beispielsweise dazu dienen, ein systemunabhängiges Interface für die Speicherverwaltung

bereitzustellen. Daher werden in der Analyse die obersten n Elemente des Aufrufpfades[2] betrachtet. Der Wert von n wird durch IR_Configuration::callChainLength festgelegt.

Um Heapblöcke für die Analyse zu erzeugen, werden die Bibliotheksfunktionen malloc, calloc und realloc ersetzt; das Ersetzen dieser Funktionen geschieht durch die in Abbildung 3.12 dargestellte Klasse alloclib. Die eingefügte Implementierung führt dazu, dass diese Funktionen einen Zeiger auf einen Heapblock zurückliefern. Dieser Heapblock wird durch ein *location-set* $< b, 0, 0 >$ unbekannter Größe repräsentiert. Bei b handelt es sich um eine Instanz der Klasse heapblock, die in Abbildung 3.13 abgebildet ist. Die Methode free, die zum Löschen von Heap-Objekten dient, wird während der Analyse nicht weiter betrachtet. Das liegt daran, dass die Analyse nicht vollständig flusssensitiv ist. So wird angenommen, dass es immer einen Kontrollflusspfad gibt, der eine einen Aufruf von free enthaltende Anweisung überspringt.

```
typedef struct list {
  int data;
  struct list *next;
} list;

int find(list *start, int data) {
  while(start != NULL) {
    if(start->data == data)
      return 1;
    start = start->next;
  }
  return 0;
}
```

Listing 3.7: Nutzung von Heapspeicher

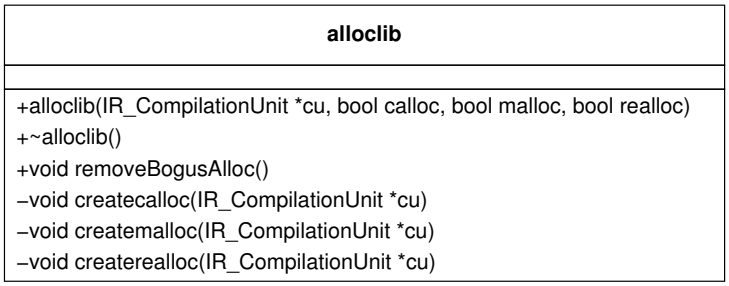


Abbildung 3.12: Klasse alloclib

[2]Dieser Pfad enthält von der Hauptfunktion aus alle zum gegenwärtigen Zeitpunkt der Ausführung aufgerufene Funktionen in Aufrufsreihenfolge.

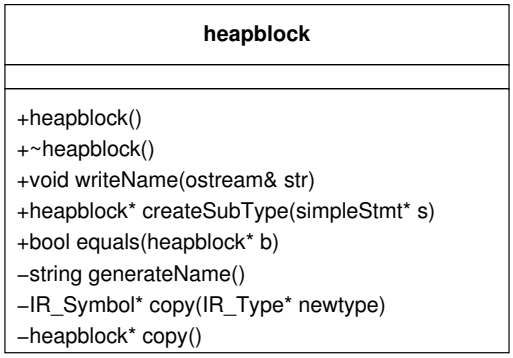


Abbildung 3.13: Klasse heapblock

Heap-Objekte werden immer als Eingabevariablen von Funktionen betrachtet, da sie auch nach Aufruf der Funktion bestehen bleiben; so werden Zuweisungen von Adressen von Heap-Objekten in die *summary* einer Funktion übernommen. Da der Aufrufpfad einer Funktion während der statischen Analyse unbekannt ist, wird er invers aufgebaut; dies geschieht beim Einfügen einer *summary* in die aufrufende Funktion: der Name der aufgerufenen Funktion wird dem Aufrufpfad von jedem Heapblock in der *summary* an erster Stelle hinzugefügt, sofern die spezifizierte Länge nicht überschritten ist. Diese Funktionalität wird von der Methode heapblock::createSubType bereitgestellt. Anschließend werden alle Blöcke mit identischem Aufrufpfad verschmolzen, d.h. der erwähnte künstliche Alias erzeugt.

3.3.9 Bibliotheken

Für Funktionen, die aus Bibliotheken zum compilierten Programmcode hinzugelinkt werden, sowie bei der Verwendung von shared libraries[3] enthält der Programmquelltext nur die Prototypen der in der Bibliothek implementierten Funktionen. Aus diesen Definitionen können keine Auswirkungen auf eine aufrufende Funktion ermittelt werden. Um dennoch eine korrekte *summary* für diese Funktionen erstellen zu können, ist eine gleichnamige Funktion mit identischer Parameterliste und gleichem Rückgabewert zu definieren, die den für die aufrufende Funktion sichtbaren Datenfluss emuliert. So entspricht die *summary* dieser Funktion der der originalen Funktion. Dabei können natürlich die bei der Erzeugung einer *summary* gemachten konservativen Annahmen genutzt werden, um den Datenfluss zu vereinfachen. Als Beispiel ist in Listing 3.8 eine Implementierung einiger Standardfunktionen aufgestellt, deren *summary* ausreichend konservativ ist. Dabei ist auch der globale Variablen einschließende Datenfluss zu beachten. Dies ist bei der gezeigten Funktion fclose von Bedeutung. Im Beispiel ist auch zu sehen, wie die in der Aliasanalyse gemachten Annahmen genutzt werden können, um die Funktionen zu vereinfachen. So sind aufgrund nicht vollständiger Flusssensitivität die aufeinan-

[3] Diese Bibliotheken werden erst zur Laufzeit verlinkt.

derfolgenden return-Anweisungen ein gültige Vereinfachung, da die jeweils letzte return-Anweisung nicht als unerreichbarer Code behandelt wird. Ähnliches gilt etwa für die Zuweisungen zu errno; die Aliasanalyse nimmt an, dass diese Variable jeden Ganzzahlwert enthalten kann.

Ist für einen Funktionsaufruf kein Funktionskörper - und damit keine *summary* - vorhanden, so müssen einer intraprozeduralen Analyse entsprechende Annahmen gemacht werden. Diese Annahmen sind in Kapitel 2.2.2 erwähnt. Um diese konservativen *Points-to*-Werte zu berechnen, werden zunächst alle als unbekannt geltenden *location-sets* in eine Liste aufgenommen. Für jedes *location-set* wird der Parameter *stride* auf den Wert 1 gesetzt und alle überlappenden Sets im *Points-to*-Graph vereinigt. Das dient zur Abschätzung von möglicher Adressarithmetik in den unbekannten Funktionen. Ferner wird angenommen, dass jeder unbekannte Speicherbereich jeden beliebigen Wert als Konstante annehmen kann.

```
char *strchr(const char *s, int c) {
  char *p = s;
  p++;
  return p;
  return 0;
}


int errno;

int fclose(FILE *stream) {
  errno=1;
  errno++;
  return 0;
  return EOF;
}


void exit(int status) {
}
```

Listing 3.8: Bibliotheksfunktionen

Zunächst werden alle globalen Variablen und alle durch einfaches Dereferenzieren der Argumente unbekannter Funktionensaufrufe erreichbaren Variablen als unbekannt betrachtet. Durch das Ändern der *stride*-Werte kann sich möglicherweise das *Points-to*-Set eines als unbekannt geltenden *location-sets* vergrößern. Anschließend werden alle Elemente aus dem *Points-to*-Set der Listenelemente ebenfalls in die Liste aufgenommen. Diese Schritte werden solange wiederholt, bis die Menge der in der Liste gespeicherten *location-sets* einen Fixpunkt erreicht hat.
Für die jetzt errechnete Menge von *location-sets* werden die *Points-to*-Werte nun derart konservativ abgeschätzt, dass jedes *location-set* auf alle *location-sets* dieser

Menge zeigt. Durch diese Vorgehensweise werden zunächst alle Speicherbereiche gesammelt, auf die die unbekannte Funktion überhaupt zugreifen kann. Anschließend werden alle denkbaren *Points-to*-Beziehungen auf dieser Menge berechnet.

Kapitel 4

Anwendungen

In diesem Kapitel wird gezeigt, wie einzelne Optimierungsalgorithmen eines Compilers die von der Aliasanalyse bereitgestellten Informationen nutzen können. Der Vorteil, den die Aliasanalyse bringt, wird mit einem Vergleich zu einer Optimierung ohne Kenntnis von Aliassen quantifiziert. Um aussagekräftige Ergebnisse zu erreichen, wurden reale Programme bzw. Algorithmen geprüft. Diese Programme sollen den Fokus auf eingebettete Systeme wiederspiegeln, was die Dominanz von Programmen zur Signalverarbeitung und Datenkompression erklärt; diese Anwendungen stellen typische performancekritische Elemente für eingebettete Systeme dar.

Programm	Beschreibung	Codezeilen
fftlib [Gre]	Bibliothek mit Funktionen für die schnelle Fouriertransformation	3693
adpcm (enc., dec.) [LPMS97]	Sprachkompression nach G.711, G.721 und G.723	1579 bzw. 1573
huffmann (enc., dec.) [Bou]	Kompressionsalgorithmen	355 bzw. 281
cavity [BTC89]	Bildverarbeitung	269
gif2asc [Fer]	Grafikprogramm	494
8051mkr [Giv]	Simulator für i8051	487
radix [WF00]	Routing von Netzwerkpaketen mit Radix-Bäumen	1558

Tabelle 4.1: Benchmarkprogramme

Für diese Programme wurde auch die Laufzeit und der Speicherverbrauch der Aliasanalyse gemessen, so dass die Verbesserungen der Programmoptimierung in Relation zum Aufwand gesetzt werden können. Diese Ergebnisse wurden auf einem System mit einem Sun Sparcv9 Prozessor mit 750Mhz und 4Gbyte Speicher ermittelt. Abbildung 4.1 zeigt die Laufzeit der Analyse für jedes der Programme, Abbildung 4.2 den benötigten Speicher.

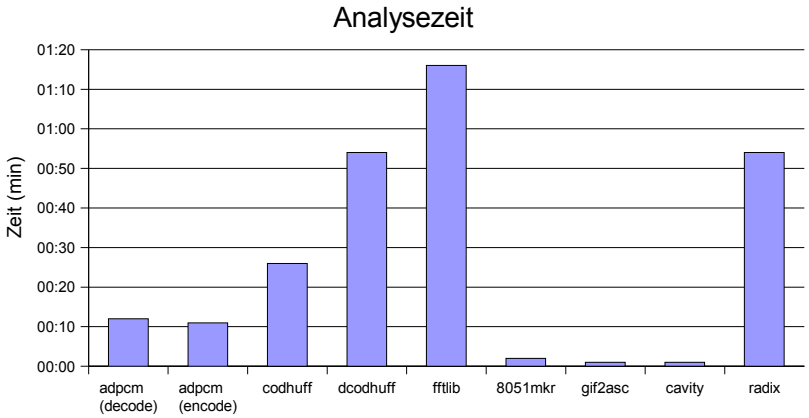


Abbildung 4.1: Laufzeit der Aliasanalyse

Diese Werte beziehen sich auf das Einlesen und Analysieren des Programms, d.h., Diagramm 4.1 enthält auch die vom Parser benötigte Zeit, und Abbildung 4.2 auch den für das *ICD-C* sowie die *Highlevel*-Zwischendarstellung benötigten Speicherplatz. Diese Ergebnisse lassen es realistisch erscheinen, die Aliasanalyse in einem Compiler innerhalb einer höheren Optimierungsstufe - z.B. zum Erzeugen von endgültigen Programmversionen - anzubieten. Lediglich der fftlib-Benchmark hat einen auffällig hohen Ressourcenbedarf; zu erklären ist dies durch sehr große Funktionen, was zu einer Iteration über sehr viele Anweisungen führt, obwohl diese Menge von Anweisungen durch die Flusssensitivität während des Erzeugens einer *summary* bereits verkleinert wird.

4.1 Registerallokation

Die Registerallokation findet im Backend eines Compilers statt. Dieses Backend wandelt eine Zwischendarstellung in Maschinencode um. In dieser Phase werden die Codeselektion und die Registerallokation durchgeführt, teilweise werden auch noch maschinenabhängige Optimierungen durchgeführt. Bei der Codeselektion wird für die in der Zwischendarstellung vorhandenen Ausdrucksbäume und Anweisungen eine Menge von Maschinenanweisungen berechnet; diese Menge wird nach gewissen Kriterien wie z.B. minimaler Ausführungszeit ausgewählt. Dazu dienen Algorithmen wie z.B. *tree-pattern-matching* [Muc97]. Müssen Werte zwischengespeichert werden, so werden temporäre Variablen erzeugt. Die Registerallokation kann nun entweder davon ausgehen, dass alle Variablen im Speicher abgelegt sind, oder aber dass diese Variablen sogenannte virtuelle Register darstellen. In beiden Fällen sind Kopieroperationen zwischen physikalischen Registern und

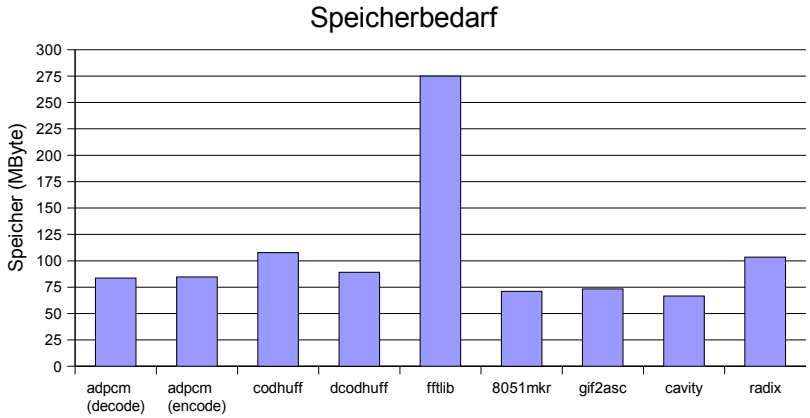


Abbildung 4.2: Benötigter Speicherplatz

Speicher einzufügen (Register-spilling), da die Maschineninstruktionen bei vielen Prozessorarchitekturen (insbesondere RISC-Prozessoren) häufig nur auf Daten in physikalischen Registern, nicht aber im Hauptspeicher zugreifen können, im Allgemeinen aber nicht genügend physikalische Register vorhanden sind, um alle Variablen zu speichern. Ziel der Registerallokation ist nun, mit möglichst wenigen Kopieroperationen auszukommen. Dabei muss eine Registerbelegung die folgenden Bedingungen erfüllen:

- Ein Register darf zu einem Zeitpunkt immer nur einer einzigen Variablen entsprechen.
- Wird eine neue Variable in ein Register kopiert, so ist der alte Wert des Registers in den Speicher zurückzuschreiben, sofern der Wert geändert wurde und an anderer Stelle im Programm weiterverwendet wird.
- Indirekte Speicherzugriffe, d.h. Dereferenzierungen von Zeigern müssen berücksichtigt werden. Bei einem indirekten Ladebefehl kann es sein, dass die Speicherstelle, auf die zugegriffen wird, in ein Register kopiert und dort verändert wurde. Vor einem Ladebefehl sind also alle Register zurückzuschreiben, die eine Variable im *Points-to*-Set des dereferenzierten Zeigers darstellen und deren Wert durch Schreibzugriffe geändert wurde. Bei einem indirekten Schreibzugriff kann es sein, dass eine Speicherstelle modifiziert wird, die bereits vorher in ein Register kopiert wurde. Der Wert eines solchen Registers ist anschließend ungültig. Wurde der Registerwert vor dem

indirekten Schreibzugriff geändert, so ist dieser vor dem indirekten Schreibzugriff in den Speicher zu kopieren. Das liegt daran, dass die durch den indirekten Schreibzugriff beschriebene Speicherstelle einen Alias mit mehreren Registern bilden kann; Ursache ist, dass es mehrere mögliche Zielvariablen für einen Zeiger geben kann. Beschrieben wird aber maximal eine in ein Register kopierte Variable, der Wert der anderen Register bleibt unverändert.

Das Problem, eine optimale Allokation zu finden, die obigen Bedingungen genügt, ist NP-Vollständig [Muc97]. Daher werden Heuristiken eingesetzt, die eine möglichst gute Allokation erreichen, aber wesentlich besser skalieren. Zur weiteren Performanceverbesserung wird diese Allokation nicht immer global durchgeführt, sondern z.B. nur für eine einzige Funktion. Auch die implementierte Registerallokation berechnet Registerbelegungen für jede Funktion einzeln.

Aus der dritten Bedingung lässt sich erkennen, wie die Allokation durch die Ergebnisse der Aliasanalyse verbessert werden kann; es stehen genaue Informationen darüber zur Verfügung, welche Register Variablen entsprechen, die mit der Zielvariable eines indirekten Speicherzugriffs einen Alias bilden. Ohne eine solche Analyse müssen konservativere Annahmen gemacht werden; häufig wird ein Alias für ein Register immer dann angenommen, wenn die Variable, deren Wert das Register enthält, mindestens einmal im Programm mit dem Adressoperator angesprochen wird (*Address-taken*-Analyse).

Die implementierte Registerallokation stützt sich zunächst auf von einem Codeselektor erzeugten Code; dieser Code nutzt den in Tabelle 4.2 dargestellten Pseudobefehlssatz, der eine RISC-Maschine modelliert. Dieser Befehlssatz besitzt Operationen zum Kopieren von Werten aus dem Speicher in ein Register sowie das Schreiben von Registerwerten in den Speicher. Alle anderen Instruktionen können nur mit Immediate- sowie Registeroperanden arbeiten. Der Codeselector nutzt die im *ICD-C* vorhandenen Funktionen für das *tree pattern matching*. Diese Funktionen erzeugen für jede Anweisung einen Datenflussbaum, dessen Knoten der Klasse IR_TreeElem jeweils eine Instanz von IR_Exp repräsentieren. Dabei ist ein Zeiger auf den zugehörigen Ausdruck gespeichert und kann mittels der Funktion `IR_Exp* IR_TreeElem::getExp()` abgefragt werden. So kann von einem IR_TreeElem auf die Methode `set<IR_Exp*>& IR_Exp::getPointsTo()` des zugehörigen Ausdrucks zugegriffen werden. Dies zeigt, wie die durch die Aliasanalyse gewonnen Informationen auch im Compilerbackend verwendet werden können.

Die Registerallokation wird für jede Funktion einzeln durchgeführt und nimmt an, dass alle in den Instruktionen verwendeten Daten virtuelle Register darstellen. Dazu wird zunächst die *live-range*[1] von Variablen berechnet. Dies geschieht durch

[1] Die *live-range* bezeichnet einen Programmteil, in dem der Wert einer Variablen genutzt wird. Dieser Teil beginnt beim ersten Schreibzugriff auf diese Variable und endet beim letzten lesenden Zugriff.

Instruktion	Beschreibung
$li\ R_1, I$	$R_1 \leftarrow I$
$ld\ R_1, I$	$R_1 \leftarrow [I]$
$lr\ R_1, R_2$	$R_1 \leftarrow [R_2]$
$sd\ R_1, I$	$[I] \leftarrow R_2$
$sr\ R_1, R_2$	$[R_1] \leftarrow R_2$
$mov\ R_1, R_2$	$R_1 \leftarrow R_2$
$jmp\ I$	$PC \leftarrow I$
$jmp.z\ R_1, I$	$R_1 = 0 :\ PC \leftarrow I$
$jmp.nz\ R_1, I$	$R_1 \neq 0 :\ PC \leftarrow I$
$test\ R_1$	Beschreibt C entsprechend R_1
$sel\ R_1, R_2, R_3$	$zero :\ R_1 \leftarrow R_3;\ \overline{zero} :\ R_1 \leftarrow R_2$
ret	$IP \leftarrow [SP--]$
$callr\ R_1$	$[++SP] \leftarrow PC; PC \leftarrow R_1$
$call\ I$	$[++SP] \leftarrow PC; PC \leftarrow I$
$push\ R_1, R_2$	$[++R_1] \leftarrow R_2$
$pop\ R_1, R_2$	$R_1 \leftarrow [R_2--]$
$add\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 + R_3$
$and\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 \wedge R_3$
$div\ R_1, R_2, R_3$	$R_1 \leftarrow \frac{R_2}{R_3}$
$mod\ R_1, R_2, R_3$	$R_1 \leftarrow R_2\ mod\ R_3$
$mul\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 * R_3$
$neq\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 \neq R_3$
$or\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 \vee R_3$
$shl\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 * 2^{R_3}$
$shr\ R_1, R_2, R_3$	$R_1 \leftarrow \frac{R_2}{2^{R_3}}$
$sub\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 - R_3$
$xor\ R_1, R_2, R_3$	$R_1 \leftarrow R_2 \oplus R_3$
$not\ R_1, R_2$	$R_1 \leftarrow \overline{R_2}$

Tabelle 4.2: RISC-Befehlssatz. $R_1 \ldots R_n$ stellen Register dar, C ein Condition-Flag mit Einträgen für *zero* und *carry*, die von arithmetischen Operationen und test gesetzt werden; I ist ein Immediate-Operand. $[x]$ kennzeichnet einen Speicherzugriff auf die in x enthaltene Adresse, PC den Instruktionszeiger. Es gibt einen Stackzeiger, der SP genannt ist.

einen iterativen Durchlauf durch die vom Backend erzeugten Maschineninstruktionen; diese Instruktionen liefern Informationen, welche Variablen beschrieben und gelesen werden. Aus diesen Informationen lässt sich ähnlich dem in Kapitel 2.2.4 vorgestellten Algorithmus zur flusssensitiven Aliasanalyse für jeden Programmpunkt die Menge der Variablen berechnen, die zu diesem Zeitpunkt *live* sind. Anschließend wird ein Graph erzeugt, der für jedes virtuelle Register einen Knoten enthält. Überlappen sich die *live-ranges* von zwei Variablen, wird eine Kante in den Graph eingefügt zwischen den beiden Knoten, die diese beiden Variablen repräsentieren. Eine solche Kante besagt, dass diese beiden virtuellen Register nicht auf das gleiche physikalische Register abgebildet werden können. Daraus folgt natürlich, dass keine gültige Registerallokation berechnet werden kann, wenn von einem Knoten n mindestens k Kanten ausgehen und k die Anzahl der der physikalischen Register ist. Solange es solche Knoten gibt, wird ein virtuelles Register gewählt, das nicht mehr auf ein physikalisches Register abgebildet werden soll. In der Implementierung wird das virtuelle Register gewählt, von dem im Graph die meisten Kanten ausgehen, es wären aber auch andere Auswahlkriterien denkbar. Da das gewählte virtuelle Register r nicht mehr in ein physikalisches Register kopiert werden soll, sind natürlich die Maschineninstruktionen abzuändern; so sind alle Vorkommen des Registers r gegen ein jeweils neues, temporäres, virtuelles Register r' auszutauschen. Wurde r in der ursprünglichen Maschineninstruktion gelesen, so ist vor dieser Instruktion der Wert von r' aus dem Speicher zu lesen; wurde r beschrieben, ist der Wert von r' in den Speicher zu schreiben. Durch diese Programmmodifikationen verkürzen sich die *live-ranges* der virtuellen Register, was dazu führt, dass der anschließend neu gebildete Graph, der Kanten für Überlappungen dieser *live-ranges* enthält, keine Knoten mit einem Grad von mindestens k mehr besitzt. Daraus folgt, dass dieser Graph mit k Farben eingefärbt werden kann. Dazu wird ein Greedy-Algorithmus verwendet, der die Knoten im Graph in beliebiger Reihenfolge abarbeitet und ihnen die minimal mögliche Farbe zuweist; dies ist die kleinste Farbe, die nicht bereits ein Nachbarknoten besitzt.

Wird eine Instruktion abgearbeitet, die einen indirekten Lesezugriff darstellt, werden erst alle modifizierten Register, die einen Alias mit der gelesenen Variable bilden können, in den Speicher zurückgeschrieben. Erst dann wird die eigentliche Instruktion abgearbeitet. Bei einem indirekten Schreibzugriff werden ebenfalls alle Register, die einen Alias mit der Zielvariablen bilden können und die beschrieben wurden, in den Speicher kopiert. Anschließend werden alle Register, die einen Alias mit der Zielvariablen bilden können als unbelegt markiert; wird ein Register weiterhin verwendet, wird der Wert nach dem indirekten Schreibzugriff wieder aus dem Speicher gelesen. Erst dann wird die eigentliche Instruktion abgearbeitet. Diese Programmveränderungen werden durch das Einfügen von zusätzlichen Kopieroperationen realisiert.

Um Vergleichswerte für eine Registerallokation ohne Aliasanalyse zu erhalten, ist auch ein Algorithmus implementiert, der für indirekte Speicherzugriffe die Menge

der zu speichernden bzw. löschenden Register dadurch bestimmt, ob diese Register den Inhalt einer bekannten Speicheradresse repräsentieren.

Ergebnisse

Für die Benchmarks wurde die Zahl der eingefügten Kopieroperationen gemessen; eine geringere Zahl gilt dabei als besser. Für die Messung wurden verschiedene Registersatzgrößen angenommen, um mehrere Prozessorarchitekturen abbilden zu können. Ermittelt wurde die Anzahl der Kopieroperationen für verschiedene Registersätze von 8 bis 256 Registern; diese Werte sollen die derzeit erhältlichen Prozessoren reflektieren. Dieser Registersatz ist homogen, daher sind die einzelnen Register beliebig austauschbar. Die eingesparten Kopieroperationen schlagen sich direkt in der Laufzeit der Programme nieder. Auf eine direkte Messung der Laufzeit wurde jedoch verzichtet, da die Ergebnisse aufgrund der abstrakten Natur der Hardwareplattform wenig aussagekräftig wären. Wie in Abbildung 4.3 zu sehen ist, bringt die Aliasanalyse für diese Anwendung abhängig vom Eingabeprogramm

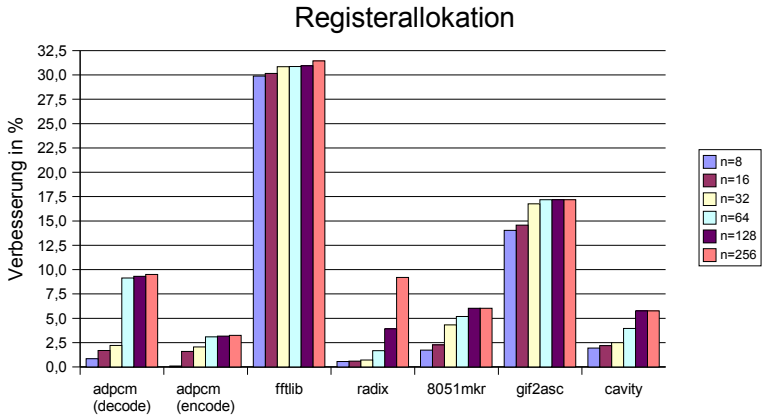


Abbildung 4.3: Ergebnisse Registerallokation

sehr unterschiedliche Ergebnisse. Dargestellt sind die prozentualen Einsparungen von Kopieroperationen; der Basiswert 100% entspricht einer Allokation ohne Berücksichtigung von Aliasinformationen.

Während der fftlib-Benchmark und gif2asc mit sehr guten Ergebnissen auffallen, zeigen andere Benchmarks teilweise nur geringe Verbesserungen. Dies tritt insbesondere bei kleinen Registersätzen auf; eine genauere Untersuchung der Ergebnisse zeigt, dass bei diesen Programmen die beschränkte Anzahl von Registern für

viele der eingefügten Kopieroperationen verantwortlich ist. Mit größeren Registersätzen dagegen wurden teilweise deutlich bessere Ergebnisse erzielt, wie es auch im dargestellten Diagramm erkennbar ist.

Dies ist darauf zurückzuführen, dass mehr Variablen in Registern bereitgehalten werden können; somit verringert sich der Anteil der durch die begrenzte Anzahl von Registern ausgelösten Kopieroperationen. Da zu erwarten ist, dass die Geschwindigkeit von Prozessoren auch in Zukunft stärker steigt als die Speicherbandbreite, ist davon auszugehen, dass die Registersätze von Prozessoren vergrößert werden. Dies würde die Bedeutung der Aliasanalyse für eine möglichst optimale Registerallokation steigern.

4.2 Static memory allocation

Ziel der *static memory allocation* ist, die im Programm deklarierten globalen Variablen im Speicher so anzuordnen, dass der Speicherverbrauch minimal ist. Um dieses Ziel zu erreichen, werden mehrere Variablen an der gleichen Stelle im Speicher abgelegt, sofern sich die *live-ranges* dieser Variablen nicht überlappen. Dies ähnelt der Registerallokation, bei der ebenfalls mehrere Variablen an einem einzigen Speicherplatz - in diesem Fall ein Register - abgelegt werden. Bei der Bestimmung der *live-ranges* werden auch indirekte Speicherzugriffe berücksichtigt. Ohne eine Aliasanalyse müsste angenommen werden, dass beim Dereferenzieren eines Zeigers auf jede globale Variable zugegriffen werden kann, die im Programm mit dem Adressoperator angesprochen wird. Mit einer Aliasanalyse ist es möglich, die Menge der Variablen auf die ermittelten möglichen Zielwerte des dereferenzierten Zeigers zu beschränken. Dies führt zu kürzeren *live-ranges* für die einzelnen Variablen und folglich zu weniger Überlappungen dieser *live-ranges*. Dadurch gibt es mehr Freiheitsgrade bei der Anordnung der Variablen im Speicher.

Der implementierte Algorithmus basiert auf dem in [Zhu01] vorgestellten. Dieser Algorithmus arbeitet auf einem Graph, der für alle globalen Variablen jeweils einen Knoten enthält. Überlappen sich die *live-ranges* von zwei Variablen, so wird eine Kante zwischen den entsprechenden Knoten eingefügt. Anschließend wird dieser Graph gefärbt. Dafür wird ein Greedy-Algorithmus verwendet; solange es noch nicht gefärbte Knoten gibt, wird einem solchen Knoten die minimale Farbe zugewiesen. Dabei handelt es sich um die kleinste Farbe, die nicht auch ein Nachbarknoten besitzt.

Um den erwähnten Graph aufzubauen, ist das gesamte Programm zu analysieren. Dies geschieht wie bei der *Bottom-up*-Phase der Aliasanalyse auf einem topologisch sortierten Funktionsaufrufgraphen von den Funktionen ausgehend, die keine weiteren Funktionsaufrufe enthalten. Das impliziert natürlich auch, dass rekursive Zyklen gesondert gehandhabt werden müssen.

Ziel bei der Analyse einer Funktion ist, die sich überlappenden *live-ranges* von Variablen zu bestimmen. Ferner sind Informationen zu bestimmen, die für aufrufende Funktionen benötigt werden. Das ist die Menge von Variablen M_1, die während der Funktionsausführung definitiv beschrieben werden, die Menge aller gelesenen Variablen M_2 sowie die Menge aller Variablen, auf die möglicherweise zugegriffen wird M_3. Die Mengen M_2 und M_3 enthalten alle Speicherzugriffe auf globale Variablen, auch solche, die durch Dereferenzieren von Zeigern entstehen. M_1 dagegen enthält nur Schreibzugriffe, die definitiv stattfinden. Das können entweder direkte Speicherzugriffe sein oder aber Dereferenzierungen von Zeigern, die nur eine Zielvariable besitzen.

Aufgrund der *Bottom-up*-Analyse wird angenommen, dass diese Werte für alle aufgerufenen Funktionen bereits berechnet sind. Es wird für jeden Basisblock b in der Funktion die Menge $live(b)$ definiert, die die Variablen enthält, deren *live-range* den Beginn des Basisblocks überlappt. Anschließend werden alle Basisblöcke mindestens einmal analysiert, bis alle *live*-Mengen einen Fixpunkt erreichen. Dabei wird während der Analyse eines Basisblocks b an jedem Programmpunkt in diesem Block das aktuelle *live-set* gebildet. Zunächst ergibt es sich aus

$$\bigcup_{b' \in succs(b)} live(b')$$

das mögliche *live-set* am Ende des Basisblocks. $succs(b)$ ist die Menge aller möglichen nachfolgenden Basisblöcke von b. Anschließend wird b entgegen der Ausführungsrichtung durchlaufen. Zunächst werden für jede Anweisung die Mengen M_1', M_2' und M_3' gebildet, deren Bedeutung analog zu den Mengen für Funktionen sind. So werden im Falle eines Funktionsaufrufs auch die Mengen M_1 bis M_3 in die Mengen M_1' bis M_3' kopiert. Dies gilt allerdings nur eingeschränkt für indirekte Funktionsaufrufe. Für indirekte Aufrufe mit mehreren möglichen Zielfunktionen werden die Mengen M_2 und M_3 zwar für jede mögliche Funktion kopiert, M_1 allerdings nicht. Das liegt daran, dass nicht klar ist, ob eine Funktion tatsächlich aufgerufen wird. Damit gibt es auch keine definitiv auftretenden Schreibzugriffe. Sowohl bei direkten als auch bei indirekten Funktionsaufrufen wird aber die Menge der möglicherweise aufgerufenen Funktionen festgehalten.

Der Ablauf der Iteration über die Basisblöcke gestaltet sich so, dass der Kontrollfluss entgegen der Ausführungsrichtung abgearbeitet wird. Es wird jeweils eine einzelne Anweisung betrachtet und die sich durch Ausführung dieser Anweisung ergebenden Veränderungen auf das *live-set* berechnet.
Zunächst werden die interprozedural überlappenden *live-ranges* gebildet. Dazu wird die Menge A gebildet, die alle Zugriffe durch aufgerufene Funktionen enthält. Sei B das aktuelle *live-set*. Nun wird für alle Kombinationen $a \in A$, $b \in B$ eine Kante zwischen a und b zum Graph hinzugefügt.

Anschließend wird das *live-set* für den Programmpunkt vor Ausführung der jeweiligen analysierten Anweisung berechnet. Dazu werden zunächst die definitiv beschriebenen Variablen aus dem *live-set* gelöscht; diese Variablen sind direkt vor einem Schreibzugriff ja ungenutzt. Das erklärt auch die konservative Handhabung der Menge M_1; ein nur möglicherweise stattfindender Schreibzugriff könnte dazu führen, dass der vorherige Wert der Variablen noch in anderer Stelle im Programm verwendet wird. Anschließend werden alle gelesenen Variablen in dieses Set eingefügt, da angenommen wird, dass diese an vorhergehender Stelle im Programm beschrieben wurden. Daraufhin wird im *live-set A* für alle $a_1 \in A, a_2 \in A,\ a_1 \neq a_2$ eine Kante zwischen a_1 und a_2 im Graph hinzugefügt.

Die Bildung der Menge M_1 geschieht durch die Definition von Mengen *written* für jeden Basisblock. Zunächst sind diese Mengen leer. Dargestellt wird die Menge von Variablen, die ab diesem Programmpunkt definitiv vor dem Beenden der Funktion beschrieben werden. Gebildet wird die Menge aus allen im Basisblock enthaltenen, definitiv auftretenden Schreibzugriffen sowie der Schnittmenge der *written*-Mengen aller nachfolgenden Basisblöcke. Eine Leseoperation löscht eine Variable aus der Menge *written*. Die Menge M_1 ist dann identisch zu der Menge *written* des Basisblocks, mit dem die Funktionsausführung beginnt.

Rekursive Zyklen werden konservativer gehandhabt. Die Menge M_1 wird immer als leer betrachtet, während die Mengen M_2 und M_3 die Zugriffe aller Funktionen im Zyklus sowie von allen aufgerufenen Funktionen, die nicht Teil dieses Zyklus sind, enthalten. Anschließend werden im Graph für alle $m_1 \in M_3, m_2 \in M_3,\ m_1 \neq m_2$ eine Kante zwischen m_1 und m_2 im Graph eingefügt. Es wird angenommen, dass die berechneten Mengen M_1 bis M_3 für alle Funktionen in diesem Zyklus gültig sind.

Um Vergleiche zu einer Allokation ohne Kenntnis von Aliasbeziehungen ziehen zu können, ist auch ein Algorithmus mit konservativeren Annahmen implementiert:

- Durch indirekte Speicherzugriffe kann auf alle globalen Variablen zugegriffen werden, deren Adresse bekannt ist; das sind die Variablen, die mit dem Adressoperator angesprochen werden.
- Indirekte Funktionsaufrufe werden konservativer gehandhabt, da die Zielfunktionen nicht bekannt sind. Da ohne genaue Kenntnisse über den Kontrollfluss unklar ist, welche globalen Variablen keine nicht-überlappenden *live-ranges* haben, wird angenommen, dass keine Variablen an der gleichen Adresse gespeichert werden dürfen.

Ergebnisse

Aus dem gefärbten Graph kann nun abgelesen werden, dass die Variablen, deren Knoten die gleiche Farbe haben, an identischen Stellen im Speicher abgelegt wer-

den können. Abbildung 4.4 zeigt die möglichen Einsparungen von Speicherplatz für die gewählten Benchmarkprogramme; die im Diagramm dargestellten Einsparungen von Speicherplatz beziehen sich auf die Allokation ohne Aliasanalyse und gehen davon aus, dass Variablen an beliebigen Stellen im Speicher abgelegt werden können, es also kein zu berücksichtigendes *Alignment* gibt. Es ist zu erkennen, dass teilweise sehr große Verbesserungen auftreten. Dies gilt insbesondere für die ADPCM-Benchmarks, was unter anderem auf die in diesen Programmen vorhandenen indirekten Funktionsaufrufe zurückzuführen ist. Die sehr großen Verbesserungen für cavity ergeben sich aus global deklarierten Vektoren, die aber wie eine *address-taken*-Variable zu behandeln sind, da sie ohne vollständige Indizierung an Unterfunktionen als Argumente übergeben werden.

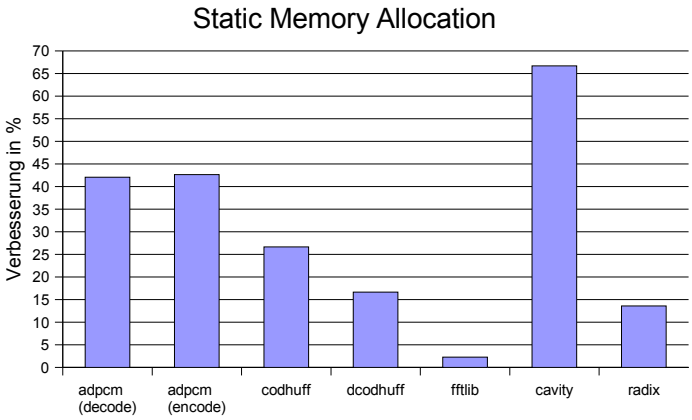


Abbildung 4.4: Ergebnisse Static Memory Allokation

4.3 Common subexpression elimination

Ziel der *common subexpression elimination* ist, in einem Programm wiederholt berechnete Ausdrücke durch nur einen Ausdruck zu ersetzen, der in einer temporären Variable zwischengespeichert wird. Alle folgenden Ausdrücke können dann durch diese Variable ersetzt werden. Dies ist allerdings nur möglich, wenn diese Ausdrücke mit identischen Datenwerten arbeiten.
Der implementierte Algorithmus basiert auf einer bereits im *ICD-C* implementierten *common subexpression elimination*, die jedoch keine Aliasinformationen verwendet. Das führt dazu, dass einige Ausdrücke, die ersetzt werden können, unbe-

rücksichtigt bleiben. Grund dafür ist, dass ohne Aliasinformationen bei Dereferenzierungen angenommen werden muss, dass ein indirekter Schreibzugriff die Werte aller Variablen ändern kann, deren Adresse bekannt ist; das sind die Variablen, die mit dem Adressoperator angesprochen werden. Das kann zu der Annahme führen, dass durch einen indirekten Schreibzugriff eine Quellvariable eines Ausdrucks beschrieben wird. Sobald jedoch der Wert von mindestens einer Quellvariablen geändert ist, kann ein zu einem vorhergehenden Ausdruck identischer Ausdruck einen anderen Wert liefern; die zwischengespeicherte Variable wird also ungültig. Mit den Ergebnissen der Aliasanalyse kann nun die Menge der durch indirekte Schreibzugriffe veränderten Variablen genauer abgeschätzt werden. Dadurch werden weniger Werte von Ausdrücken, die in temporären Variablen gespeichert sind, als ungültig markiert. Folglich kann eine größere Anzahl von Ausdrücken ersetzt werden.

Ergebnisse

Um die Vorteile der Aliasanalyse quantifizieren zu können, wurden die ausgewählten Benchmarkprogramme mit beiden Varianten der *common subexpression elimination* optimiert und die Anzahl der jeweils ersetzten gemeinsamen Teilausdrücken verglichen. Die Ergebnisse sind in Abbildung 4.5 zu sehen; dort ist dargestellt, wie viele zusätzliche Teilausdrücke durch Nutzung von Aliasinformationen entfernt werden konnten. Abgebildet sind dabei prozentuale Verbesserungen verglichen mit einer Optimierung ohne Aliasinformationen. Eine Geschwindigkeitssteigerung der Programme kann sich daraus ergeben, dass das Zwischenspeichern eines Ausdrucks weniger zeitaufwendig ist als eine Neuberechnung. Durch die für einen Ausdruck dem Programm hinzugefügte temporäre Variable kann es allerdings dazu kommen, dass sich an einigen Programmpunkten die Menge der Variablen, die *live* sind, vergrößert. Das kann dazu führen, dass das Programm häufiger mit im Hauptspeicher vorgehaltenen Datenwerten arbeitet. Insbesondere, wenn nur einfache Ausdrücke ersetzt werden, kann es dadurch vorkommen, dass sich die Laufzeit sogar verlängert. Ob dieser Fall eintritt, hängt allerdings von der Zielplattform ab. Relevante Parameter sind beispielsweise die Anzahl der Register, die für einzelne Ausdrücke benötigte Rechenzeit sowie die Speicherzugriffszeit. Da die implementierte *common subexpression elimination* auf einer plattformunabhängigen *Highlevel*-Darstellung arbeitet werden die genannten Einschränkungen der Zielplattform nicht berücksichtigt. Die Ergebnisse zeigen entsprechend die maximale Menge von Ausdrücken auf, die ersetzt werden können.

Insbesondere für die ADPCM-Programme, aber auch für den Huffmann-Encoder und den Routing-Algorithmus ergeben sich sehr gute Ergebnisse. Für den fftlib-Benchmark ergibt sich nur eine geringe Verbesserung. Zwar wird eine große Zahl von gemeinsamen Teilausdrücken erkannt, dabei handelt es sich aber weitgehend um arithmetische Berechnungen mit Variablen, die nicht mit dem Adressoperator angesprochen werden; ebenso werden während dieser Berechnungen keine weiteren Unterfunktionen aufgerufen, die einen Effekt auf diese Variablen haben könn-

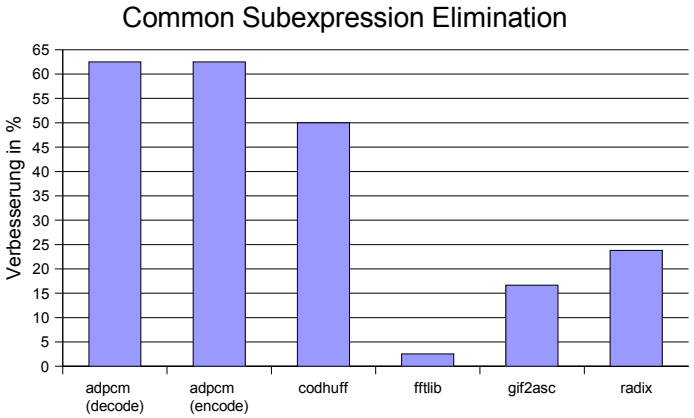


Abbildung 4.5: Ergebnisse Common Subexpression Elimination

ten. So kann auch ohne eine Aliasanalyse erkannt werden, dass es keine indirekten Schreibzugriffe auf die jeweiligen Quellvariablen der Ausdrücke geben kann.

4.4 Partial redundancy elimination

Bei der *partial redundancy elimination* handelt es sich um eine Erweiterung der *common subexpression elimination*. Während letztere Optimierungsmethode nur definitiv bereits berechnete Ausdrücke ersetzt, betrachtet die *partial redundancy elimination* auch Ausdrücke, die nur möglicherweise bereits berechnet wurden. Ein solcher Fall ist in Abbildung 4.6 dargestellt. Der dort dargestellte Programmausschnitt besteht aus drei Basisblöcken A, B und C. Es existiert ein Kontrollflusspfad von A und von B nach C. Ferner gibt es einen Ausdruck exp, der in A und in C berechnet wird. Es wird angenommen, dass die erste Berechnung von exp in Basisblock A und C nicht durch andere Ausdrücke gelöscht wird, bevor der Ausdruck in Basisblock C neu berechnet wird. Folglich ist die Neuberechnung von exp nicht erforderlich, wenn der Basisblock C über A erreicht wird. Eine *common subexpression elimination* kann diesen Ausdruck aber nicht entfernen, da bei Erreichen von C über B dieser Ausdruck nicht berechnet wurde. Während der *partial redundancy elimination* wird daher die Berechnung von exp als letzte Anweisung von B hinzugefügt und in einer temporären Variablen gespeichert; dies geschieht auch mit dem in Basisblock A gespeicherten Ausdruck. So kann der Ausdruck in C durch die temporäre Variable ersetzt werden.

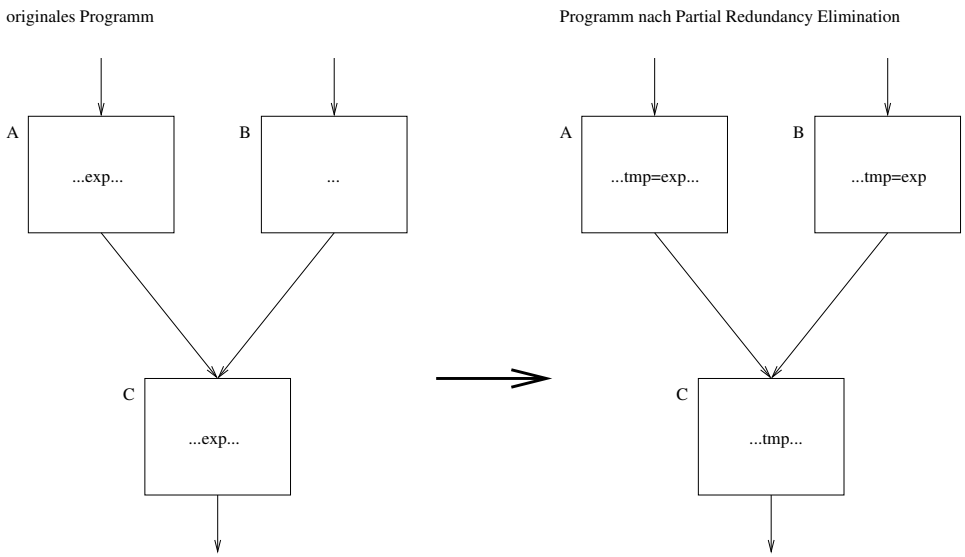


Abbildung 4.6: Beispiel zur Partial Redundancy Elimination

Allgemein läuft der Algorithmus [KRS92] in mehreren Schritten ab. Zunächst wird für jeden Basisblock eine lokale *common subexpression elimination* durchgeführt. Daher können redundante Ausdrücke innerhalb eines Basisblocks nicht mehr auftreten. Anschließend werden für jeden Basisblock b die folgenden Mengen von Ausdrücken berechnet:

- **Transparency** Enthält alle Ausdrücke e, für die keine Anweisung in b eine Eingabevariable ändert.
- **Local availability** Enthält Ausdruck e, wenn dieser Ausdruck in b berechnet wird und in nachfolgenden Anweisungen keine Quellvariable von e geändert wird.
- **Local anticipability** Enthält alle Ausdrücke e, die in b berechnet werden und das gleiche Ergebnis liefern würden, wenn die Berechnung zu Beginn des Basisblocks stattfinden würde.

Mit Hilfe dieser Mengen können für jeden Basisblock b die folgenden Mengen von Ausdrücken berechnet werden:

- **Global availability** Berechnet werden die Mengen der am Anfang und am Ende eines Basisblocks verfügbaren Ausdrücke, avIn und avOut genannt. avIn ist die Schnittmenge der avOut-Mengen aller vorhergehenden Basisblöcke von b, avOut für einen Basisblock ist die Menge der Ausdrücke aus avIn, die auch in Transparency enthalten sind vereinigt mit der Menge Local availability.

- **Global Anticipability** Es wird für einen Basisblock berechnet, ob ein Ausdruck am Ende dieses Basisblocks evaluiert werden kann; die Menge dieser Ausdrücke wird antOut genannt. Definiert wird ebenfalls die Menge der Ausdrücke, die am Anfang des Basisblocks ausgewertet werden kann; diese Menge wird antIn genannt. antOut ist die Schnittmenge der antIn-Mengen aller b nachfolgenden Basisblöcke. antIn selbst ist die Vereinigung aus Local anticipability sowie der Schnittmenge von Local transparency und antOut des jeweiligen Basisblocks.

avOut enthält also alle nach Ausführung eines bestimmten Basisblocks verfügbaren Ausdrücke, antIn des Basisblocks b alle Ausdrücke die, wenn zu Beginn von b berechnet, den gleichen Wert haben wie an ihrer ursprünglichen Position. antOut ist äquivalent, enthält aber die Ausdrücke, die nach Ausführung der Anweisungen in b berechnet werden können.

Im nächsten Schritt werden aus diesen Informationen die Punkte im Kontrollflussgraph bestimmt, die die frühestmögliche Position bestimmen, an der ein Ausdruck berechnet werden kann. Bei diesen Punkten handelt es sich um Kanten im Kontrollflussgraph, ein Punkt ist also eine Kombination von zwei Basisblocken (p, b). Berechnet wird die Menge earliest für alle Kombinationen (p, b), die im Kontrollflussgraph auftreten. Diese Menge enthält alle Ausdrücke, die an (p, b), nicht aber einer vorhergehenden Position berechnet werden können. Ist p der Funktionseintrittsblock, so ist dies für einen Ausdruck e genau dann der Fall, wenn die Menge antIn von b e enthält, d.h., eine Berechnung an dieser Position überhaupt möglich ist. Zusätzlich muss gelten, dass der Ausdruck nicht nach Ausführung von p zur Verfügung steht, d.h., die Menge avOut von p e nicht enthält. Handelt es sich bei p nicht um einen Funktionseintrittsblock, so ist zusätzlich zu prüfen, ob der Ausdruck nicht durch p "durchgereicht" werden kann. Ausdruck e kann durchgereicht werden, wenn er in der Menge Local transparency von p enthalten ist und am Anfang aller nachfolgenden Basisblöcke evaluiert werden kann.

Werden nun an den in earliest spezifizierten Punkten die jeweiligen Ausdrücke ausgewertet und einer temporären Variablen zugeordnet, so werden alle bisherigen Ausdrücke redundant, die sich in einer Menge für Local anticipability befinden; diese Ausdrücke können durch die jeweiligen temporären Variablen ersetzt werden. Da die Menge earliest so berechnet wird, dass nie zwei Ausdrücke dieser Menge redundant sein können ist sichergestellt, dass das Programm keine redundanten Ausdrücke mehr enthält. Daher können auch nicht zusätzliche redundante Ausdrücke eingefügt werden. Das führt dazu, dass kein Ausführungspfad verlängert wird; dies gilt allerdings nur für die compilerinterne Zwischendarstellung. Wie bei der *common subexpression elimination* kann es allerdings auch bei dieser Optimierung zu einer längeren Ausführungszeit durch zusätzliches Register-spilling kommen; eine möglichst gute Optimierung ist ebenfalls nur unter Berücksichtigung von Informationen über die Zielplattform möglich.

Zwar kann die *partial redundancy elimination* eine größere Menge von Ausdrücken als redundant erkennen als die *common subexpression elimination*, jedoch können die zusätzlich eingefügten Operationen den Programmcode vergrößern. Daher hat letztere Optimierungsmethode weiterhin ihre Berechtigung.

Ergebnisse

Gemessen wurde die Anzahl der als redundant erkannten Ausdrücke. Diese Messung wurde einmal mit einer Optimierung, die sich auf Aliasinformationen stützt, durchgeführt, ein anderes Mal wurden indirekte Speicherzugriffe ungenauer über die *address-taken*-Eigenschaft abgeschätzt. Dargestellt in Abbildung 4.7 ist die relative Verbesserung durch Nutzung der Aliasanalyse gegenüber der *address-taken*-Analyse. Der Vergleich der einzelnen optimierten Programme untereinander zeigt, dass die Ergebnisse ähnlich zu denen der *common subexpression elimination* sind, absolut gesehen liegen die Werte jedoch auf einem deutlich höheren Niveau (siehe Anhang). Bezüglich der Laufzeit entspricht die Bedeutung der Ergebnisse der der *common subexpression elimination*, ein Laufzeitvorteil ergibt sich allerdings nur, wenn die Ausführung einem Pfad folgt, auf dem es im nicht optimierten Programm redundante Ausdrücke gab.

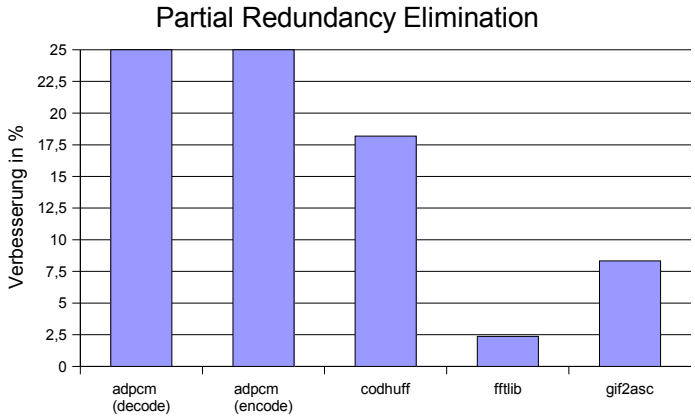


Abbildung 4.7: Ergebnisse Partial Redundancy Elimination

4.5 Weitere Anwendungsmöglichkeiten

In diesem Abschnitt werden weitere Optimierungsmöglichkeiten vorgestellt, die von den Ergebnissen der Aliasanalyse profitieren können. Diese Optimierungen werden aus theoretischer Sicht betrachtet.

Instruction scheduling

Die im Compiler-Backend vom Codegenerator erzeugten Maschinenbefehle sind möglicherweise in einer nicht optimalen Reihenfolge. Dies kann dazu führen, dass bei der Ausführung ein sogenannter Pipeline-Stall auftritt. Das ist der Fall, wenn für eine Instruktion nicht alle Operanden zur Verfügung stehen oder benötigte Hardwarekomponenten im Prozessor von anderen Instruktionen belegt sind. Insbesondere bei superskalaren Prozessoren, d.h. Prozessoren mit mehreren Pipelines, kann dieser Fall sehr häufig eintreten. Daher wird häufig versucht, mit Techniken wie *scoreboarding* oder mit dem Tomasulo-Algorithmus [HP03] im Prozessor die Instruktionen zur Laufzeit umzuordnen. Da diese Techniken jedoch Chipfläche und Energie verbrauchen, ist es erstrebenswert, das *instruction scheduling* bereits im Compiler durchzuführen. VLIW- bzw. EPIC-Architekturen verzichten gar vollständig auf ein hardwaregestütztes *instruction scheduling*.

Bei der Umordnung der Instruktionen darf natürlich die Semantik des Programms nicht verändert werden. Daher sind Datenabhängigkeiten zwischen den einzelnen Instruktionen zu berücksichtigen. Diese Abhängigkeiten lassen sich in drei Klassen einteilen:

1. **write-after-write**
   Die Reihenfolge von zwei Schreibzugriffen auf die gleiche Speicherstelle darf nicht verändert werden.

2. **write-after-read**
   Wird eine Speicherstelle gelesen und anschließend beschrieben, so dürfen diese Speicherzugriffe nicht vertauscht werden; sonst würde der falsche Wert aus dem Speicher gelesen.

3. **read-after-write**
   Wird eine Speicherstelle beschrieben und anschließend gelesen, darf die Leseoperation nicht vor die Schreiboperation verschoben werden. Dies würde dazu führen, dass ein alter Wert aus dem Speicher gelesen würde.

Auf Ebene der Maschineninstruktionen sind für Dereferenzierungen von Zeigern Schreib- bzw. Leseoperationen eingefügt. Durch eine Aliasanalyse kann die Menge der möglicherweise beschriebenen bzw. gelesenen Speicherstellen abgeschätzt werden. Ohne eine solche Analyse müsste angenommen werden, dass alle Speicherstellen beschrieben oder gelesen werden können. Damit kann durch Kenntnis

der Aliasbeziehungen die Menge der möglicherweise beschriebenen bzw. gelesenen Speicherstellen stark verkleinert werden. Um beim *instruction-scheduling* korrekte Werte zu erzeugen, wird angenommen, dass auf jede Speicherstelle in dieser Menge zugegriffen wird. Daher gibt es mit einer Aliasanalyse wesentlich weniger der oben erwähnten Datenabhängigkeiten. Dies führt zu größeren Freiheitsgraden bei der Umordnung von Instruktionen, was die Ausführungsgeschwindigkeit des Programmcodes erhöhen kann.

Cache-conscious memory allocation

Die immer größer werdenden Geschwindigkeitsunterschiede zwischen Prozessoren und Speicherbausteinen haben zur Verwendung von Caches geführt. Das sind kleine, schnelle Speicher, die häufig verwendete Daten beinhalten sollen. Um den Hardwareaufwand für diese Caches zu begrenzen, sind diese häufig nicht vollassoziativ ausgeführt, sondern besitzen k Cachezeilen mit je m Speicherplätzen für Cacheblöcke; ein solcher Cache wird m-fach assoziativ genannt. Das bedeutet, dass ein Speicherbereich aus dem Hauptspeicher nur an eine begrenzte Menge an Stellen im Cache kopiert werden kann. Die Menge der Positionen ergibt sich dabei aus der Adresse, an dem der Datenblock im Hauptspeicher abgelegt ist. Üblicherweise wird die Adresse direkt auf eine Cachezeile abgebildet. So kann es vorkommen, dass es in einem Programmteil eine Menge von n häufig benutzten Speicherbereichen gibt, die an insgesamt nur m verschiedenen Positionen gespeichert werden können. Ist $m < n$, so verdrängen sich diese Datenblöcke gegenseitig aus dem Cache. Dieses Problem wird auch als *thrashing* bezeichnet. Um es zu vermeiden, werden durch diese Optimierung Variablen im Speicher so angeordnet, dass diese für jeden Programmteil in maximal m Cacheblöcke passen oder aber die $n > m$ Cacheblöcke, die benötigt werden, nicht nur an m Stellen im Cache gespeichert werden können [Muc97] [CCJA98]. Neben Informationen zum Aufbau der in der Hardware verwendeten Caches ist auch die Menge der in einem Programmabschnitt verwendeten Variablen möglichst exakt zu bestimmen. Die Aliasanalyse kann dabei helfen, die Menge der Speicherstellen, auf die durch Dereferenzierungen zugegriffen wird, genauer zu bestimmen.

Schleifenoptimierungen

Zu dieser Gruppe gehören eine Vielzahl von Optimierungen [Muc97]. Ziel dieser Optimierungen ist die Verbesserung der Ausführungsgeschwindigkeit von Schleifen. Da Schleifen einen großen Anteil an der Ausführungszeit von Programmen haben, sind diese Optimierungen von großer Bedeutung. Zu dieser Gruppe von Optimierungen gehören z.B. das “Abrollen” von Schleifen, Vertauschen von geschachtelten Schleifen oder eine Veränderung der Ausführungsreihenfolge der Iterationen einer Schleife. Die Optimierung einer Schleife ist allerdings oft an einige Bedingungen geknüpft. Zu diesen Bedingungen kann beispielsweise gehören, dass

der Datenfluss einer Induktionsvariable vollständig bekannt sein muss. Dies ist bei indirekten Speicherzugriffen möglicherweise nicht der Fall. Durch die Ergebnisse einer Aliasanalyse kann möglicherweise sichergestellt werden, dass durch einen indirekten Speicherzugriff nicht auf die Induktionsvariable zugegriffen wird. Aber auch für andere Variablen können die Aliasbeziehungen ausgenutzt werden. So verringert sich die Menge der in Kapitel 4.5 genannten Datenabhängigkeiten. Diese Abhängigkeiten bestehen auch zwischen zwei Iterationen der Schleife. Fallen einige solcher Abhängigkeiten weg, so kann es dadurch mehr Möglichkeiten geben, die Schleifeniterationen umzuordnen. Durch die Aliasanalyse kann also die Menge der Schleifen, die optimiert werden kann, vergrößert werden.

Globale Programmoptimierungen

Globale Programmoptimierungen nutzen für ihre Ergebnisse Informationen aus dem gesamten Programmquelltext. Dies steht im Gegensatz zu nur lokal durchgeführten Optimierungen, die sich z.B. auf eine Funktion beschränken. Werden Optimierungen für jede Funktion einzeln durchgeführt, so ist es möglich, dass die Optimierung einer Funktion das Optimierungspotential einer anderen Funktion verringert. Dies kann dazu führen, dass für ein Programm als Ganzes keine optimale Form gefunden wird. Globale Optimierungen versuchen, diese Fälle zu vermeiden. Dafür ist natürlich eine möglichst genaue Kenntnis des Funktionsaufrufgraphen unabdingbar. Für Programme mit indirekten Funktionsaufrufen ist dieser Graph nicht vollständig bekannt, daher wird die Menge der durch einen indirekten Funktionsaufruf ausgeführten Funktionen konservativ abgeschätzt. So werden interprozedurale Abhängigkeiten berücksichtigt, die bei der Programmausführung gar nicht entstehen. Mit einer Aliasanalyse dagegen können Zielwerte von Funktionszeigern besser abgeschätzt werden, was die Menge der Abhängigkeiten reduziert. Dies eröffnet größere Freiräume für die Optimierung.

Kapitel 5

Zusammenfassung und Ausblick

Dieses Kapitel fasst die durchgeführten Arbeiten zusammen und gibt einen Ausblick auf die weiteren Anwendungs- und Verbesserungsmöglichkeiten der Aliasanalyse.

5.1 Zusammenfassung

Hauptziel dieser Arbeit war die Entwicklung einer Aliasanalyse für *C*-Programme. Um diese Analyse möglichst vielseitig verwenden zu können, arbeitet sie auf einer *Highlevel*-Darstellung der Eingabeprogramme, die vollständig plattformunabhängig ist. Damit auch die Analyseergebnissse nicht plattformabhängig sind, kann die Analyse auf eine bestimmte Zielplattform konfiguriert werden; dies geschieht durch die Definition von Datentypgrößen und Funktionsaufrufkonventionen.

Zunächst wurden verschiedene Analysetechniken miteinander verglichen, die sich bezüglich der Komplexität sowie der Präzision der Ergebnisse teilweise deutlich unterscheiden. Auf diesen Vergleichen basieren letztlich die Entwurfsentscheidungen, die sich darauf konzentrieren, Aliasbeziehungen mit möglichst großer Genauigkeit zu berechnen. So ist die implementierte Analyse als *field-* und *context-sensitive* zu klassifizieren. Dazu wurde die Analyse in zwei alternierende Phasen aufgeteilt. In einer Phase wird für jede Funktion berechnet, welche Auswirkungen die Ausführung dieser Funktion auf die aufrufende Funktion hat; diese Auswirkungen werden in einer *summary* dargestellt. In der anderen Phase werden die eigentlichen Aliasbeziehungen berechnet, statt Funktionsaufrufen werden dabei die jeweiligen *summaries* betrachtet; so sind interprozedurale Effekte in dieser Phase nicht zu berücksichtigen. Um *field-sensitivity* zu erreichen, arbeitet die Analyse mit *location-sets*, mit denen auch mehrdeutige Zugriffe innerhalb von einem Speicherbereich dargestellt werden können; dabei kann ein *location-set* durch die Definition einer Schrittweite unendlich viele Stellen in einem Speicherbereich repräsentieren; so können alle Zugriffe mit einer beschränkten Zahl von *location-sets* ausgedrückt werden. Um die *location-sets* möglichst genau zu bestimmen, werden

auch arithmetische Ausdrücke im Programm analysiert, da diese der Adressberechnung dienen können. Damit baut die Analyse auf dem in [NKH04a] vorgestellten Algorithmus auf, ist allerdings vollständig *field-sensitive*, nutzt Kontrollflussinformationen zur Berechnung der *summaries* und bietet weitergehende Möglichkeiten für die Analyse von Daten, die auf dem Heap abgelegt werden.

Anschließend wurde gezeigt, wie bestehende Optimierungsalgorithmen in einem Compiler von den Ergebnissen der Aliasanalyse profitieren können. Dazu wurden verschiedene Optimierungsalgorithmen implementiert; Ergebnisse wurden sowohl mit als auch ohne Verwendung von Aliasinformationen berechnet. Der Vergleich zeigt, dass die Nutzung von Aliasinformationen teilweise zu sehr großen Vorteilen führen kann; bei der Common Subexpression bzw. Redundancy Elimination beispielsweise konnte die Anzahl der ersetzten Teilausdrücke um bis zu 62,5% vergrößert werden, während bei der Static Memory Allocation bis zu 66,7% Speicherplatz eingespart werden konnten. Bei der Registerallokation konnte die Zahl der benötigten Kopieroperationen um bis zu 31,4% gesenkt werden.

5.2 Ausblick

Die in Kapitel 4 implementierten Techniken stellen nur einen Bruchteil aller Optimierungen dar, die von Aliasinformationen profitieren können. Weitere Möglichkeiten zur Programmoptimierung sind dort bereits aufgeführt, jedoch erhebt diese Aufstellung nicht den Anspruch auf Vollständigkeit. Für das *ICD-C* bieten sich natürlich insbesondere die vorgestellten *Highlevel*-Optimierungen wie beispielsweise Schleifenoptimierungen an.

Verbesserungspotenzial gibt es noch bei der Geschwindigkeit der Analyse; so wäre es möglicherweise vorteilhaft, in der *Top-down*-Phase (Kapitel 3.3.7) die *Points-to*-Werte für ein *location-set* zwischenzuspeichern; derzeit kann es vorkommen, dass diese Werte wiederholt auf gleicher Datenbasis berechnet werden. Ähnliches gilt für die in einem Speicherbereich möglicherweise abgelegten Konstanten. Für diese Verbesserung wäre allerdings festzustellen, auf welche *location-sets* eine Zuweisung einen Einfluss haben kann (z.B. durch eine Zuweisungskette). Dies könnte beispielsweise dadurch geschehen, dass der *Points-to*-Graph in seine einzelnen Zusammenhangskomponenten unterteilt wird; diese Komponenten bilden sich aus PTGNode-Instanzen, als Kanten sind sowohl die baseElements- als auch pointsTo-Kanten zu berücksichtigen. Mit dieser Einteilung in einzelne Komponenten könnte die Menge der *location-sets* eingeschränkt werden, auf die eine Zuweisung eine Auswirkung hat; so wären nicht bei jeder Zuweisung alle berechneten und gespeicherten Zwischenergebnisse zu invalidieren.

Weiterhin könnte es möglich sein, vor der Codeumwandlung zu analysieren, welche Programmteile zur Berechnung von Speicheradressen dienen. Da alle anderen

Programmteile für die Berechnung der Aliasinformationen irrelevant sind, könnten die zu diesen Programmabschnitten gehörenden Anweisungen weggelassen werden.

Eine weitere Möglichkeit wäre die bereits in Kapitel 3.3.3 angedeutete Kombination mit einer Optimierung zur *function specialization*, durch die Rekursionen genauer analysiert werden könnten. Diese *function specialization* könnte prinzipiell sogar völlig unabhängig von der Aliasanalyse arbeiten, d.h. die Aliasanalyse würde nach der *function-specialization* ausgeführt werden. Dagegen spricht jedoch, dass auch die *function-specialization* die Ergebnisse der Aliasanalyse nutzen kann, beispielsweise um indirekte Funktionsaufrufe genauer betrachten zu können.

Die Analyse von Heap-Blöcken bietet ebenfalls Raum für Erweiterungen; so stellen die Aufrufpfade nur eine relativ grobe Abschätzung dar, welche Blöcke zu einer Datenstruktur gehören. Eine genauere Analysemöglichkeit wird beispielsweise in [LA03] vorgestellt. Diese Form der Analyse versucht, Operationen auf einzelnen Datenstrukturen zu analysieren. Um einzelne Datenstrukturen auseinanderzuhalten, wird ein sogenannter *data structure graph* aufgebaut, der alle Operationen sowie Speicherbereiche als Knoten enthält; greift eine Operation auf einen Speicherbereich zu, wird eine Kante zwischen beiden Komponenten eingefügt. Anschließend stellen die einzelnen Zusammenhangskomponenten in diesem Graphen einzelne Datenstrukturen dar.

Anhang A

Ergebnisse

Laufzeit der Aliasanalyse

Programm	Laufzeit in Minuten
ADPCM (decode)	0:12
ADPCM (encode)	0:11
huffmann (decode)	0:54
huffmann (encode)	0:26
8051mkr	0:02
gif2asc	0:01
fft	1:16
radix	0:54
cavity	0:01

Tabelle A.1: Laufzeit der Aliasanalyse

Benötigter Speicher für die Aliasanalyse

Programm	Speicher in MByte
ADPCM (decode)	84
ADPCM (encode)	85
huffmann (decode)	89
huffmann (encode)	108
8051mkr	71
gif2asc	73
fft	275
radix	103
cavity	67

Tabelle A.2: Speicherverbrauch der Aliasanalyse

	Anzahl der Register					
Programm	8	16	32	64	128	256
ADPCM (decode)	953	771	587	460	451	442
ADPCM (encode)	932	748	586	387	378	369
fft	11885	11776	11512	11512	11512	11449
radix	2141	2018	1670	1445	1174	1174
8051mkr	289	220	162	135	116	116
gif2asc	791	762	662	646	646	646
cavity	412	364	320	202	139	139

Tabelle A.3: Anzahl der eingefügten Kopieroperationen ohne Aliasanalyse

	Anzahl der Register					
Programm	8	16	32	64	128	256
ADPCM (decode)	945	758	574	418	409	400
ADPCM (encode)	931	736	574	375	366	357
fft	8333	8224	7960	7958	7950	7849
radix	2129	2006	1658	1412	1128	1066
8051mkr	284	215	155	128	109	109
gif2asc	680	651	551	535	535	535
cavity	404	356	312	194	131	131

Tabelle A.4: Anzahl der eingefügten Kopieroperationen mit Aliasanalyse

Static memory allocation

Programm	Speicher mit Aliasanalyse	Speicher ohne Aliasanalyse
ADPCM (decode)	598	1032
ADPCM (encode)	592	1032
huffmann (decode)	20	24
huffmann (encode)	22	30
cavity	3366132	10098672
fft	344	352
radix	305	353

Tabelle A.5: Für globale Variablen allozierter Speicher in Bytes

Common subexpression elimination

Programm	mit Aliasanalyse	ohne Aliasanalyse
ADPCM (decode)	26	16
ADPCM (encode)	26	16
huffmann (encode)	12	8
gif2asc	7	6
fft	324	316
radix	26	21

Tabelle A.6: Anzahl der ersetzten gemeinsamen Teilausdrücke

Partial Redundancy Elimination

Programm	mit Aliasanalyse	ohne Aliasanalyse
ADPCM (decode)	40	32
ADPCM (encode)	40	32
huffmann (encode)	26	22
gif2asc	13	12
fft	346	338

Tabelle A.7: Anzahl der entfernten gemeinsamen Teilausdrücke

Anhang B

Dokumentation der Beispielanwendungen

Registerallokation

Das Programm *regalloc_pa* ermittelt eine mögliche Registerallokation unter Nutzung von Aliasinformationen, *regalloc_no_pa* arbeitet ohne diese Informationen. Beide Programme geben zunächst Pseudo-Maschinencode und anschließend die Zahl der als *register-spilling* eingefügten Kopieroperationen aus. Beim Aufruf spezifiziert der erste Parameter die Anzahl der Register, die restlichen Parameter definieren die Eingabedateien. Ein Aufruf hat also folgende Form:
regalloc_pa Register Datei1...DateiN
Die Parameter für das Programm *regalloc_no_pa* haben identische Bedeutung.

Static-memory allocation

Das Programm ist mit allen Quelldateien aufzurufen. Die entsprechende Kommandozeile lautet folglich
memalloc Datei1...DateiN
Als Ausgabe erscheint eine Tabelle, in der jedem global definierten Symbol ein Wert zugewiesen wird. Symbole mit identischem Wert werden nie gleichzeitig genutzt und können folglich im Speicher an der gleichen Adresse abgelegt werden. Anschließend wird aus den Größen der Variablentypen der minimal benötigte Speicherplatz ermittelt. Beide Ausgaben werden zunächst mit, anschließend ohne Verwendung von Aliasinformationen ermittelt.

Common subexpression elimination

Es stehen die Programme *cse_pa* und *cse_no_pa* zur Verfügung. Nur ersteres nutzt die Ergebnisse der Aliasanalyse. Aufzurufen sind die Programme mit
cse_pa Pfad Datei1...DateiN
beziehungsweise

cse_no_pa Pfad Datei1...DateiN
Die optimierten Quelldateien befinden sich anschließend im Unterverzeichnis *Pfad*.

Partial redundancy elimination

Die Programme *pre_pa* und *pre_no_pa* führen die *partial redundancy elimination* durch. Nur ersteres nutzt die Ergebnisse der Aliasanalyse. Aufzurufen sind die Programme mit
pre_pa Pfad Datei1...DateiN
beziehungsweise
pre_no_pa Pfad Datei1...DateiN
Die optimierten Quelldateien befinden sich anschließend im Unterverzeichnis *Pfad*.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[And94] ANDERSEN: *Program Analysis and Specialization for the C Programming Language*, University of Copenhagen, Diss., 1994

[Bou] BOURGIN, David: *Codecs*. http://hpux.connect.org.uk/hppd/hpux/Languages/codecs-1.0/

[BTC89] BISTER, M. ; TAEYMANS, Y. ; CORNELIS, J.: Automatic Segmentation of Cardiac MR Images. In: *Proceedings of "Computers in Cardiology"*. Jerusalem, Sep 1989, S. 215–218. – Ordner Paper I

[CCJA98] CALDER, B. ; CHANDRA, K. ; JOHN, S. ; AUSTIN, T.: Cache-Conscious Data Placement. In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose, 1998

[DEC] *Alias Analysis in the DEC C and DIGITAL C++ Compilers*. http://research.compaq.com/wrl/DECarchives/DTJ/DTJT04/DTJT04HM.HTM

[Fer] FERRELL, John: *Ascii Gif Viewer*. http://ftp.isu.edu.tw/pub/Windows/Chinese/txt/soft/gif2asc.asc

[GBT+05] GUO, Bolei ; BRIDGES, Matthew J. ; TRIANTAFYLLIS, Spyridon ; OTTONI, Guilherme ; RAMAN, Easwaran ; AUGUST, David I.: Practical and Accurate Low-Level Pointer Analysis. In: *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA : IEEE Computer Society, 2005, S. 291–302

[GCCa] *GCC 3.3.1*. http://www.stanford.edu/services/pubsw/package/languages/gcc.html

[GCCb] *GCC Internals - Alias analysis*. http://gcc.gnu.org/onlinedocs/gccint/Alias-analysis.html

[Giv] GIVARGIS, Tony: *Synthesizable VHDL Model of 8051*. http://www.cs.ucr.edu/ dalton/i8051/i8051syn/

[GLS01] GHIYA, Rakesh ; LAVERY, Daniel M. ; SEHR, David C.: On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In: *PLDI*, 2001, S. 47–58

[Gre] GREEN, John: *FFT library*. http://cluster.earlham.edu/detail/cairo/src/benchfft-2.0/c_source/green/

[Hin01] HIND, Michael: Pointer Analysis: Haven't We Solved This Problem Yet? In: *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*

[HP98] HIND, Michael ; PIOLI, Anthony: Assessing the Effects of Flow-Sensitivity on Pointer Alias Analysis. In: *5th International Static Analysis Symposium*, 1998

[HP03] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003

[HS02] HARBISON, Samuel P. ; STEELE, Guy L.: *C: A Reference Manual, 5th edition*. Prentice Hall, 2002

[HT01] HEINTZE, Nevin ; TARDIEU, Olivier: Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In: *SIGPLAN Conference on Programming Language Design and Implementation*, 254-263

[IBM] *IBM Compilers for pSeries*. http://www.spscicomp.org/ScicomP4/Presentations/Blainey/scicomp1001.pdf

[INT] *Intel Developer UPDATE Magazine*. http://www.intel.com/technology/magazine/computing/it03001.pdf

[KR88] KERNIGHAN, Brian W. ; RITCHIE, Dennis M.: *The C Programming Language, Second Edition, ANSI C*. Prentice Hall International, 1988

[KRS92] KNOOP, J. ; RUETHING, O. ; STEFFEN, B.: Lazy code motion. In: *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* Bd. 27. San Francisco, CA, June 1992, 224–234

[LA03] LATTNER, C. ; ADVE, V.: *Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis*. 2003

[LPMS97] LEE, Chunho ; POTKONJAK, Miodrag ; MANGIONE-SMITH, William H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In: *International Symposium on Microarchitecture*, 1997, S. 330–335

[Mar03] MARWEDEL, Peter: *Embedded System Design*. Kluwer Academic Publishers, 2003

[Muc97] MUCHNIK, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997

[NKH04a] NYSTROM, Erik M. ; KIM, Hong-Seok ; HWU, Wen-mei W.: Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In: *SAS*, 2004, S. 165–180

[NKH04b] NYSTROM, Erik M. ; KIM, Hong-Seok ; HWU, Wen-mei W.: Importance of heap specialization in pointer analysis. In: *PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM Press, 2004, S. 43–48

[Ste96] STEENSGAARD, Bjarne: Points-to analysis in almost linear time. In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM Press, 1996. – ISBN 0–89791–769–3, S. 32–41

[Weg] WEGENER, Ingo: *Grundvorlesung Datenstrukturen, Wintersemester 01/02*

[WF00] WOLF, Tilman ; FRANKLIN, Mark A.: *CommBench - A Telecommunications Benchmark for Network Processors*. Austin, TX, April 2000

[Wil97] WILSON, Robert: *Efficient, context-sensitive pointer analysis for C programs*, Stanford University, Diss., 1997

[Zhu01] ZHU, J.: Static memory allocation by pointer analysis and coloring. In: *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, IEEE Press, 2001, S. 785–790