Universität des Saarlandes

Naturwissenschaftlich-Technische Fakultät 1
Lehrstuhl für Programmiersprachen und Übersetzerbau,
Prof. Dr. Reinhard Wilhelm

Generisches Slicing auf Maschinencode

Diplomarbeit
zur Erlangung des akademischen Grades
„Diplom-Informatiker“

Marc Schlickling
schlickling@cs.uni-sb.de

Februar 2005

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Danksagung

Bei Herrn Prof. Dr. Reinhard Wilhelm möchte ich mich für die Vergabe dieses interessanten Themas sowie für seine Hilfsbereitschaft und Unterstützung bedanken. Herrn Prof. Dr. Bernd Finkbeiner danke ich für die Bereitschaft, diese Arbeit zu begutachten. Darüberhinaus möchte ich mich bei Herrn Dr.-Ing. Florian Martin für die intensive Betreuung, die vielen hilfreichen Diskussionen und das Korrekturlesen dieser Arbeit bedanken. Dafür möchte ich mich ebenfalls bei Herrn Dr. Reinhold Heckmann bedanken. Mein Dank gilt weiterhin meinem Freund und Kollegen Markus Pister für die vielen hilfreichen Diskussionen und Hinweise. Herrn Dr.-Ing. Stephan Thesing danke ich für die Unterstützung und zahlreichen nützlichen Hinweise. Nicht zuletzt danke ich allen Mitarbeitern von AbsInt Angewandte Informatik GmbH, ohne deren Know-How diese Arbeit nicht zu verwirklichen gewesen wäre.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Komplexität von Softwaresystemen hat in den letzten Jahren stetig zugenommen. Die Anzahl verschiedener Rechnerarchitekturen, verschiedener Betriebssysteme, unterschiedlicher Mikroprozessoren und der Bereich der eingebetteten Systeme - die Vielzahl der spezifischen Eigenschaften dieser Komponenten oder Kombinationen dieser ist von einem einzigen Entwickler nicht mehr beherrschbar. Zunehmende Kooperationen von Firmen leisten durch gemeinsame Entwicklungsarbeit dazu ihr übriges.

Betrachtet man beispielsweise die Kraftfahrzeugindustrie, dann wird bei der Vielzahl unterschiedlicher Fahrzeugsysteme, die in Kraftfahrzeugen der neuesten Generation eingesetzt werden, die Entwicklung der Software nicht mehr nur von einem einzigen Unternehmen geleistet. Diese Systeme entstehen heute durch die Zusammenarbeit von immer mehr Zuliefererfirmen und Kooperationspartnern. Doch was passiert, wenn die unterschiedlichen Komponenten, die beispielsweise zur Motorsteuerung verwendet werden, nicht korrekt zusammenarbeiten?

In diesen Fällen kann Programm-Slicing Antworten liefern, die von einem Programmierer nur noch mühsam erbracht werden können. Fragestellung wie z.B. “Woher kommt der Wert dieser Variable?” sind in einem komplexen System, an dem mehrere Entwickler oder gar Entwicklerteams arbeiten, nur noch schwer von einem Programmierer zu beantworten.

Die Fragestellungen, zu deren Beantwortung Programm-Slicing herangezogen werden kann, sind vielfältig und nicht auf eine bestimmte Phase im Entwicklungsprozess beschränkt. So können die Techniken des Slicings auch oder gerade zum nachträglichen Programmverständnis eingesetzt werden. Auch die Sprachebene, auf der das Programm-Slicing eingesetzt werden soll, stellt keine Einschränkung dar. Die Konzepte des Slicings können auf viele verschiedene Programmiersprachen und Abstraktionen angepasst werden.

Ziel dieser Diplomarbeit ist es, einen generischen Algorithmus für das Slicing von Programmen zu entwickeln, der auf der Disassemblerebene von Rechnerarchitekturen arbeitet. Dazu stehen Werkzeuge zur Verfügung, die ein ausführbares Programm

in eine geeignete Zwischendarstellung überführen. Auf Basis dieser, den Kontrollfluss des Programms beschreibenden, Zwischendarstellung müssen dazu zunächst die Abhängigkeiten der verwendeten Register, sowie der einzelnen Instruktionen untereinander berechnet werden. Dabei soll auf möglichst wenig prozessorspezifische Informationen zurückgegriffen werden. Es liegt daher nahe, diese Berechnung direkt auf einer abstrakten Semantik - unabhängig von der konkreten Semantik eines Prozessors - durchzuführen. Zur effizienten Modellierung von Speicherzugriffen wird eine dynamische Methode vorgestellt, die die Ergebnisse einer externen Werte-Analyse verwendet. Der auf diesen Vorberechnungen aufbauende Slicing-Algorithmus ist, soweit mir dies zur Zeit bekannt ist, der erste generische Algorithmus zur Berechnung von Slices auf Maschinencode.

Die vorliegende Arbeit ist wie folgt gegliedert: Kapitel 2 soll einen grundsätzlichen Überblick über verschiedene Ziele und Techniken des Programm-Slicings bieten. Kapitel 3 soll dann einige theoretische Grundlagen für die darauf folgenden Kapitel aufzeigen. In Kapitel 4 wird ein Überblick über die Ziele und Techniken der globalen Datenflussanalyse gegeben. Hier werden einige grundlegende Algorithmen zur Lösung von Datenflussproblemen gezeigt und zudem der Begriff der “Abstrakten Interpretation” eingeführt. Kapitel 5 beschäftigt sich ausgiebig mit einer Berechnungsmethode des statischen Rückwärts-Slicing. Anschließend werden Datenflussprobleme formuliert, mit deren Hilfe sich für das Slicing benötigte Informationen, wie z.B. die Datenabhängigkeit, berechnen lassen. Kapitel 6 präsentiert eine dynamische Datenstruktur, mit deren Hilfe sich Speicherzugriffe in einer Datenflussanalyse effizient modellieren lassen. In Kapitel 7 wird dann die Implementierung der vorgestellten Analysen gezeigt. Die Nutzbarkeit des so entstandenen Slicing-Werkzeug wird anschließend anhand einiger Testprogramme gezeigt und die Ergebnisse vorgestellt. In Kapitel 8 werden dann die Erkenntnisse dieser Arbeit überblicksartig zusammengefasst und ein Resümee gezogen. Im Anhang dieser Arbeit finden sich einige ausgewählte Testprogramme.

Kapitel 2

Programm-Slicing

Die Fehlersuche in Programmen erfordert immer mehr Aufwand. Dies wird zum einen durch immer kürzere Entwicklungszyklen, zum anderen aber auch durch Kooperationen von mehr und mehr Firmen bedingt. Um fehlerhaftes Verhalten auch in Programmen großer Komplexität zu identifizieren, muss ein Entwickler zunächst die Struktur des Programms erkennen. Darauf aufbauend kann er nach und nach die Anweisungen eines Programms in interessante und uninteressante, den Fehler betreffende, Anweisungen einteilen. Dadurch ist er in der Lage, das fehlerhafte Verhalten auf eine bestimmte Folge von Anweisungen innerhalb des Programms einzuschränken und so den Fehler zu beseitigen.

1979 wurde von Weiser eine Methode veröffentlicht, mit der ein Teil der Abstraktionen, die ein Entwickler bei der Fehlersuche im Quellcode eines Programms in Gedanken durchführt, automatisch berechnet werden kann ([Wei79], [Wei82] und [Wei84]). Er führte dazu den Begriff des Programm-Slicings ein. Ein **Slice** für einen Programmpunkt[1] besteht dabei aus allen Programmpunkten, die direkt oder indirekt einen Einfluss auf die Werte an dem betrachteten Punkt haben. Die Aufgabe, Slices für beliebige Programmpunkte zu berechnen, wird als **Programm-Slicing** bezeichnet. In den folgenden Jahren wurden viele verschiedene Definitionen für Slices und viele verschiedene Methoden zu deren Berechnung vorgestellt.

Weiser definierte einen Slice als ein reduziertes, ausführbares Programm, das aus dem Eingabeprogramm durch Weglassen von Anweisungen entstanden ist und Teile des Verhaltens des Eingabeprogramms nachbildet. Eine andere, weit verbreitete, Definition ist, einen Slice als Teilmenge von Anweisungen und Kontrollprädikaten zu definieren, die die Werte des betrachteten Programmpunktes direkt oder indirekt beeinflussen, die aber nicht notwendigerweise ein ausführbares Programm beschreiben.

Eine wichtige Unterscheidung bei der Berechnung von Slices ist die zwischen statischem und dynamischem Slicing. Während das statische Slicing Slices für alle möglichen Eingaben eines Programms zu berechnen versucht, betrachtet das dyna-

[1] Ein Programmpunkt ist ein Punkt in einem Programm, direkt vor oder hinter einer Anweisung.

```
read(n);
int fac = 1;
int summe = 0;
while ( n >= 0 ) {
  if ( n == 0 )
    fac = fac * 1;
  else
    fac = fac * n;
  summe = summe + n;
  n--;
}
write(fac);
write(summe);
```

(a)

```
read(n);
int fac = 1;

while ( n >= 0 ) {
  if ( n == 0 )
    fac = fac * 1;
  else
    fac = fac * n;

  n--;
}
```

(b)

Listing 2.1: (a) Beispielprogramm (b) Statischer Slice bezüglich $(6, fac)$

mische Slicing lediglich eine feste Eingabe und somit eine konkrete Ausführung eines Programms.

Im folgenden wird ein Überblick über verschiedene Methoden zur Berechnung von statischen und dynamischen Slices vorgestellt.

2.1 Statisches Slicing

Beim statischen Slicing werden für die Berechnung eines Slices nur Informationen verwendet, die statisch verfügbar sind. Das bedeutet, das keine konkrete Ausführung des Programms als Basis des Slicings dient, sondern alle möglichen Ausführungen in ihrer Gesamtheit betrachtet werden müssen. Zur Berechnung eines Slices muss zunächst ein Programmpunkt sowie eine Teilmenge der Variablen, die in diesem Programmpunkt vorkommen, gewählt werden. Diese beiden Informationen werden als **Slicing-Kriterium** bezeichnet und dienen als Grundlage für die Berechnung eines Slices.

Listing 2.1(a) zeigt ein einfaches Beispielprogramm, dass die Fakultät und die Summe von 0 bis zu einer Zahl n berechnet. Betrachtet man nun einen Slice für das Slicing Kriterium $(6, fac)$, so gehören dazu alle Anweisungen, in denen die Variable fac zuletzt verändert wurde. In diesem Beispiel findet man so die Programmpunkte 2, 6 und 8. Zusätzlich gehören auch alle Anweisungen zum Slice, die die Ausführung des gewählten Programmpunktes kontrollieren, d.h. an denen eine Entscheidung getroffen werden muss. Dadurch erhält man die Programmpunkte 4 und 5. Für die so gefunde-

nen Programmpunkte wiederholt man den gerade beschriebenen Vorgang, solange, bis keine neuen Programmpunkte mehr gefunden werden. Im Beispiel sieht man, dass die Programmpunkte 4 und 5 die Variable n verwenden. Sucht man die letzte Definition dieser Variablen, so erhält man die Programmpunkte 1 und 10. Das Ergebnis dieser Berechnung ist in Listing 2.1(b) dargestellt.

Das Ergebnis dieses Vorgangs wird als statischer Slice bezüglich des gegebenen Kriteriums bezeichnet und repräsentiert die Teilmenge des Eingabeprogramms, die die Werte der gewählten Variablen im Slicing-Kriterium bestimmen. Keine andere Anweisung des Eingabeprogramms kann noch dafür in Frage kommen. Diese Eigenschaft ist besonders für die Fehlersuche in Programmen ein großer Vorteil. Mit dem Programm-Slicing bekommt der Programmierer ein Werkzeug, das ihm einen großen Teil der Programmanweisungen als uninteressant für den gesuchten Fehler kennzeichnet und ihm so einen Teil der Arbeit abnimmt. Er kann sich dann zur Fehlersuche auf die Programmpunkte beschränken, die im Slice für das gewählte Kriterium enthalten sind.

Zur Berechnung von statischen Slices existieren viele verschiedene Methoden, die teilweise auf spezielle Anforderungen und Eigenschaften der Eingabeprogramme abgestimmt sind. Eine Einteilung der Algorithmen sowie eine Zusammenfassung der möglichen Anwendungsgebiete findet sich in [Tip95].

Der erste Algorithmus zur Berechnung von statischen Slices wurde von Weiser vorgestellt. Zur Berechnung sammelt man für eine gegebene Anweisung alle Anweisungen, die eine Daten-[2] oder Kontrollabhängigkeit[3] auf die betrachtete Anweisung besitzen. Letztere werden dabei im folgenden auch als **Kontrollprädikate** bezeichnet. Betrachtet man das in Listing 2.1(a) dargestellte Beispielprogramm, so ist z.B. Anweisung 6 von Anweisung 5 kontrollabhängig, d.h. 5 ist ein Kontrollprädikat. Außerdem ist z.B. Anweisung 4 von Anweisung 1 datenabhängig.
Startet man nun vom gegebenen Slicing-Kriterium aus, so gehören alle Anweisungen zum Slice, die eine Daten- oder Kontrollabhängigkeit auf diese Anweisung besitzen. Für alle so gefundenen Anweisungen wiederholt man diesen Vorgang so lange, bis keine neuen Anweisungen mehr in die Menge des Slices aufgenommen werden.

Auf dieser Idee von Weiser aufbauend, formulierten Ottenstein und Ottenstein das Problem des Programm-Slicings als Erreichbarkeitsproblem in einem Graphen, dem sogenannten **Programm-Abhängigkeits-Graphen** ([OO84]). Dabei beschreiben die Knoten eines solchen Graphen die Anweisungen und Kontrollprädikate, die Kanten die Daten- und Kontrollabhängigkeiten. Das Slicing-Kriterium wird mittels eines Knotens identifiziert, d.h. im Unterschied zu der Methode von Weiser, wird bei diesem Verfahren der Slice immer für alle in der durch den Knoten repräsentierten Anweisung

[2]Eine Datenabhängigkeit besteht zwischen Anweisungen, in denen eine Anweisung einen Wert schreibt, den die zweite Anweisung zur Ausführung benötigt.

[3]Eine Kontrollabhängigkeit besteht zwischen zwei Anweisungen, wenn eine Anweisung bestimmt, ob die zweite Anweisung ausgeführt wird.

vorkommenden Variablen berechnet. Ein Slice hinsichtlich des gewählten Kriteriums besteht dann aus allen Knoten, von denen aus man den betrachteten Knoten erreichen kann. Abbildung 2.1 zeigt den Programm-Abhängigkeits-Graphen für das Beispiel aus Listing 2.1(a). Die Kontrollabhängigkeiten sind durch dicke, die Datenabhängigkeiten durch dünne Kanten dargestellt.

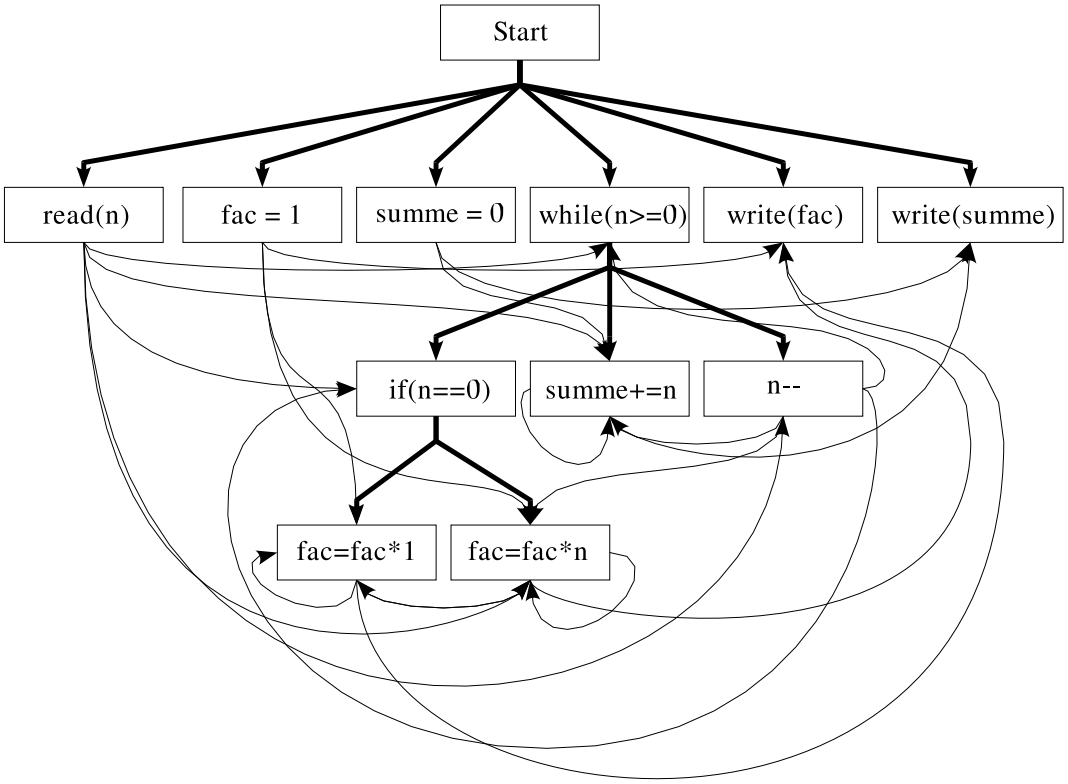


Abbildung 2.1: Programm-Abhängigkeits-Graph zu Listing 2.1(a).

Auf dem Algorithmus von Ottenstein und Ottenstein aufbauend existieren viele Erweiterungen, die diesen Ansatz auch auf komplexere Programme erweitern. Weitere Einzelheiten dazu finden sich in [Tip95].

2.2 Dynamisches Slicing

Die Grundidee des dynamischen Slicings entspricht einer Variante der *Flowback*-Analyse, die von Balzer vorgestellt wurde. Den Begriff des "dynamischen Slices" führten dann 1988 Korel und Laski ein ([KL88]). Die Fragestellung bleibt dabei die Gleiche wie bei der Analyse von Balzer: man ist interessiert daran, wie Informationen durch ein Programm hindurch propagiert werden, um so einen bestimmten Wert an einem vorher bestimmten Programmpunkt zu bilden.

```
read(n);
int fac = 1;

while ( n >= 0 ) {
  if ( n == 0 )
    fac = fac * 1;




}

```

Listing 2.2: Dynamischer Slice für $(n = 0, 6^{①}, fac)$

Der Hauptunterschied zum statischen Slicing besteht darin, dass das dynamische Slicing lediglich eine Ausführung des Eingabeprogramms betrachtet, während beim statischen Slicing alle möglichen Ausführungen des Programms betrachtet werden. Damit lassen sich Slices berechnen, die weniger Anweisungen enthalten, da zur Berechnung dieser nur eine mögliche Instanz des Programms analysiert wird.

Die Berechnung eines dynamischen Slices wird auf konkreten Eingabewerten eines Eingabeprogramms durchgeführt. Dazu ist es notwendig, den Begriff der **Belegung** einzuführen. Eine Belegung ist eine Zuordnung, die jeder externen Variable[4] einen eindeutigen Wert zuordnet. Ein Slicing-Kriterium zur Berechnung eines dynamischen Slices umfasst neben einer Belegung zusätzlich die Anweisung, für die der Slice berechnet werden soll. Dabei wird hier zwischen verschiedenen Ausführungen derselben Anweisung unterschieden, was besonders für die Betrachtung von Schleifen interessant ist. Zusätzlich muss auch zur Berechnung eines dynamischen Slices eine Teilmenge der vorkommenden Variablen der Anweisung angegeben werden, für die der Slice berechnet werden soll.

Listing 2.2 zeigt einen dynamischen Slice des Beispielprogramms aus Listing 2.1(a) für das Slicing Kriterium $(n = 0, 6^{①}, fac)$, also einen Slice für den Wert der Variablen fac während der ersten Ausführung der 6. Anweisung, wenn der Wert der Variablen $n = 0$ ist. Durch die angegebene Belegung kann in diesem Beispiel ausgeschlossen werden, dass die Schleife des Beispielprogramms bereits einmal durchlaufen wurde, womit sich die Programmpunkte, die im Programmablauf nach der betrachteten Anweisung

[4] Der Begriff "externe Variable" bezeichnet eine Variable, deren Wert zur Ausführungszeit des Programms von außerhalb in das Programm eingebracht wird.

kommen, als uninteressant eingestuft werden können.

Auch zur Berechnung von dynamischen Slices existieren viele verschiedene Algorithmen. Einzelheiten über die Vielfalt der vorgeschlagenen Lösungsansätze lassen sich in [Tip95] nachlesen.

2.3 Vorwärts-Slicing

Alle bisher vorgestellten Ansätze zur Berechnung von Slices bedienen sich eines Verfahrens, bei dem, ausgehend vom Slicing-Kriterium, rückwärts über das Eingabeprogramm gewandert und so der Slice berechnet wird. Diese Slices kann man daher auch als **Rückwärts-Slices** bezeichnen.

Die ersten, die den Begriff des **Vorwärts-Slices** einführten, waren Reps und Bricker ([RB89]). Sie benutzen dazu eine Notation, die von Bergeretti und Carré im Jahre 1985 vorgestellt wurde ([BC85]).

Ein Vorwärts-Slice besteht dabei aus allen Anweisungen und Kontrollprädikaten, die von dem gewählten Slicing-Kriterium abhängen. Abhängen bedeutet, dass diese Anweisungen vom gewählten Kriterium kontroll- oder datenabhängig sind. Dies entspricht genau der umgekehrten Abhängigkeit, die beim Rückwärts-Slicing von Interesse war.

Mit Hilfe eines Vorwärts-Slices lässt sich beispielsweise die Frage “Wenn der Wert dieser Variablen verändert wird, wie verändert sich dadurch der Programmablauf?” beantworten. Ein mögliches Anwendungsgebiet kann beispielsweise die Integration von Softwarekomponenten sein: Werden mehrere Komponenten von unterschiedlichen Entwicklern zusammengefügt, dann müssen oftmals noch kleinere Änderungen am Code vorgenommen werden. Welchen Einfluss diese Änderungen dann auf den nachfolgenden Code haben, lässt sich mittels Vorwärts-Slicing wesentlich besser abschätzen.

Listing 2.3 zeigt einen Vorwärts-Slice des Beispielprogramms aus Listing 2.1(a) für das Slicing-Kriterium $(9, summe)$.

Für die Berechnung von Vorwärts-Slices können die gleichen Algorithmen verwendet werden, die schon bei der Berechnung von Rückwärts-Slices vorgestellt wurden. Lediglich die Betrachtung der Abhängigkeiten der Anweisungen untereinander muss in der entgegengesetzten Richtung erfolgen. Betrachtet man beim Rückwärts-Slicing die Benutzung einer Variablen und “sucht” dann dazu die entsprechende Definition, so wird beim Vorwärts-Slicing die Richtung umgekehrt. Man betrachtet die Definition einer Variablen und “sucht” alle in der Programmausführung späteren benutzenden Vorkommen dieser.

Aufgrund der engen Verwandtschaft von Vorwärts- und Rückwärts-Slicing wird in dieser Arbeit nur letzteres näher untersucht. Die Konzepte können aber, wie bereits

```
2
4
6
8
    summe = summe + n;
10
12
write(summe);
```

Listing 2.3: Statischer Vorwärts Slice für $(9, summe)$

beschrieben, beliebig auf das Vorwärts-Slicing angepasst werden.

In den folgenden Kapiteln werden nun zunächst einige Grundlagen der Datenflussanalyse vorgestellt. Kapitel 5 beschreibt dann den Algorithmus von Weiser genauer und zeigt dann Datenflussprobleme, mit deren Hilfe sich dieser Algorithmus mittels abstrakter Interpretation auf die Disassemblerebene anpassen lässt.

Kapitel 3

Mathematische Grundlagen

Dieses Kapitel soll einen kurzen Überblick über die mathematischen Grundlagen der in dieser Arbeit verwendeten Konzepte bieten. Dazu werden die Grundbegriffe der Verbandstheorie und Fixpunktiteration eingeführt. Darüberhinaus werden einige Konstruktionen auf Verbänden und partiellen Ordnungen vorgestellt.

3.1 Verbandstheorie

Definition 3.1.1 (Partielle und totale Ordnung). Sei V eine Menge. Eine Relation $\sqsubseteq \subseteq V \times V$ heißt *partielle Ordnung* auf V, wenn gilt:

1. Reflexivität: $\forall v \in V : v \sqsubseteq v$
2. Transitivität: $\forall u, v, w \in V : u \sqsubseteq v \wedge v \sqsubseteq w \Rightarrow u \sqsubseteq w$
3. Antisymmetrie: $\forall v, w \in V : v \sqsubseteq w \wedge w \sqsubseteq v \Rightarrow v = w$

Die Relation heißt *totale Ordnung* auf V, wenn zusätzlich gilt:

$$\forall v, w \in V : v \sqsubseteq w \vee w \sqsubseteq v$$

Eine Menge V mit einer partiellen Ordnung $\sqsubseteq$ heißt *partiell geordnete Menge* $(V, \sqsubseteq)$. □

Definition 3.1.2 (Obere Schranke). Sei $(V, \sqsubseteq)$ eine partiell geordnete Menge und $P \subseteq V$. Ein Element $v \in V$ heißt *obere Schranke* von P, wenn gilt:

$$\forall p \in P : p \sqsubseteq v$$

Ein Element $v \in V$ heißt *kleinste obere Schranke* von P, $\bigsqcup P$, wenn gilt:

1. v ist eine obere Schranke von P,

2. für alle oberen Schranken q von P gilt: $v \sqsubseteq q$.

$\bigsqcup$ heißt *Vereinigung*. Für die kleinste obere Schranke von zwei Elementen v_1 und v_2 schreibt man auch: $v_1 \sqcup v_2$ □

Eine größte untere Schranke $\bigsqcap$ kann analog dazu definiert werden. $\bigsqcap$ heißt dann *Schnitt*.

Definition 3.1.3 (ω-Kette, vollständige partielle Ordnung). Sei $(V, \sqsubseteq)$ eine partiell geordnete Menge. Eine *ω-Kette*[1] einer partiellen Ordnung ist eine aufsteigende Kette aus Elementen von V der Form:

$$v_0 \sqsubseteq v_1 \sqsubseteq \ldots \sqsubseteq v_i \sqsubseteq \ldots$$

$(V, \sqsubseteq)$ heißt *vollständige partielle Ordnung*, wenn für jede ω-Kette $v_0 \sqsubseteq v_1 \sqsubseteq \ldots \sqsubseteq v_i \sqsubseteq \ldots$ die kleinste obere Schranke der Menge $\{v_n \mid n \in \omega\}$ existiert. □

Definition 3.1.4 (Aufsteigende Kettenbedingung). Eine ω-Kette $v_0 \sqsubseteq v_1 \sqsubseteq \ldots \sqsubseteq v_i \sqsubseteq \ldots$ erfüllt die *aufsteigende Kettenbedingung*, wenn ab einem $n \in \omega$ gilt:

$$v_n = v_{n+1} = \ldots$$

□

Definition 3.1.5 (Vollständiger Verband). Eine partiell geordnete Menge $(V, \sqsubseteq)$ heißt *vollständiger Verband*, wenn jede Teilmenge $P \subseteq V$ eine größte untere und eine kleinste obere Schranke besitzt. Einen Verband schreibt man dann häufig als Tupel $(V, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ mit $\bot = \bigsqcap V$ und $\top = \bigsqcup V$. □

Definition 3.1.6 (Dualer Verband). Sei $(V, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband. Den *dualen Verband* erhält man, indem man folgende Symbole miteinander vertauscht: $\sqsubseteq$ mit $\sqsupseteq$, $\sqcup$ mit $\sqcap$ und $\bot$ mit $\top$. □

Definition 3.1.7 (Monotone Funktion). Seien $(D, \sqsubseteq_D)$ und $(W, \sqsubseteq_W)$ zwei partiell geordnete Mengen. Eine Abbildung $f : D \to W$ heißt *monoton*, wenn gilt:

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

□

Definition 3.1.8 (Distributive Funktion). Seien $(D, \sqsubseteq_D)$ und $(W, \sqsubseteq_W)$ zwei vollständige Verbände. Eine Abbildung $f : D \to W$ heißt *distributiv*, wenn gilt:

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

□

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

Definition 3.1.9 (Stetige Funktion). Seien $(D, \sqsubseteq_D)$ und $(W, \sqsubseteq_W)$ zwei vollständig partiell geordnete Mengen. Eine Abbildung $f : D \to W$ heißt *stetig*, wenn für alle Ketten $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \ldots$ in D gilt:

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

□

3.2 Konstruktionen auf partiellen Ordnungen

Satz 3.2.1 (Kartesisches Produkt). Seien $(V_i, \sqsubseteq_i)$ für $i = 1, \ldots, n$ vollständig partiell geordnete Mengen. Das Produkt $V_1 \times V_2 \times \ldots \times V_n$ mit

$$(v_1, \ldots, v_n) \sqsubseteq (w_1, \ldots, w_n) \Leftrightarrow \forall i \in \{1, \ldots, n\} : v_i \sqsubseteq_i w_i$$

bildet wieder eine vollständig partiell geordnete Menge. Wenn für alle V_i gilt, dass es sich um vollständige Verbände handelt, dann gilt dies auch für das Produkt dieser Verbände. □

Satz 3.2.2 (Potenzverband). Sei V eine Menge. Das Tupel $(\mathcal{P}(V), \emptyset, V, \subseteq, \cup, \cap)$ bildet einen vollständigen Verband. □

Satz 3.2.3 (Funktionsbildung). Sei W eine Menge und $(V, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband. Der Funktionenraum

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

mit der Ordnung

$$f \sqsubseteq g \Leftrightarrow \forall w \in W : f(w) \sqsubseteq g(w)$$

bildet wieder einen vollständigen Verband. □

Satz 3.2.4 (Lifting). Sei $(V, \bot_V, \top_V, \sqsubseteq_V, \sqcup_V, \sqcap_V)$ ein vollständiger Verband. Dann ist $(L, \bot_L, \top_L, \sqsubseteq_L, \sqcup_L, \sqcap_L)$ mit $L = V \cup \{\bot_L, \top_L\}$ und

$$l_1 \sqsubseteq_L l_2 \Leftrightarrow \begin{cases} l_1 = \bot_L \\ l_2 = \top_L \\ l_1 \sqsubseteq_V l_2 \end{cases}$$

ein vollständiger Verband. □

Die Beweise, dass die vorgestellten Konstruktionen die geforderten Eigenschaften erfüllen, können in [Mar95] nachgelesen werden.

3.3 Fixpunktiteration

Oftmals müssen rekursive Gleichungssysteme, bei denen die zu definierenden Werte nicht nur auf der linken Seite der Gleichung vorkommen, gelöst werden. Als prominentes Beispiel kann man dazu die Fakultätsfunktion betrachten:

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

Von einer Lösung wird erwartet, dass sie die Gleichung erfüllt. Zur Lösung dieser rekursiven Definitionen kann man sich eines einfachen, iterativen Ansatzes bedienen. Dabei beginnt man mit dem kleinsten Element des Lösungsraums $\perp$ und setzt es in die Definitionsgleichung ein. Dadurch erhält man eine Definition für das nächst größere Element. Diesen Vorgang wiederholt man n mal und erhält so eine Funktion, die auf dem Intervall $[0, \ldots, n-1]$ definiert ist. Bildet man nun den Grenzwert bei $n \to \infty$, so erhält man die gesuchte Funktion auf den natürlichen Zahlen.

Definition 3.3.1 (Präfixpunkt). Sei $f : V \to V$ ein Funktion. Ein *Präfixpunkt* von f ist ein Element $v \in V$ mit $f(v) \sqsubseteq v$. □

Definition 3.3.2 (Fixpunkt). Sei $f : V \to V$ ein Funktion. Ein Element $v \in V$ heißt *Fixpunkt* von f wenn $f(v) = v$. □

Das gerade informell am Beispiel der Fakultätsfunktion vorgestellte Verfahren lässt sich in folgendem Satz mathematisch formulieren:

Satz 3.3.1 (Fixpunktiteration). Sei $(V, \sqsubseteq)$ eine vollständig partiell geordnete Menge und $f : V \to V$ eine stetige Funktion.
Sei $\mathit{fix}(f) = \bigsqcup_{n \in \omega} f^n(\perp)$. Dann ist $\mathit{fix}(f)$ ein Fixpunkt von f und der kleinste Präfixpunkt von f. Es gilt:

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

□

Dieser Satz lässt sich recht einfach beweisen:

Beweis 3.3.1. Zum Beweis müssen die beiden Teilaussagen einzeln bewiesen werden:

zu 1.

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

zu 2. Sei d ein Präfixpunkt. Es gilt, dass $\bot \sqsubseteq d$. Durch die Monotonie folgt, dass $f(\bot) \sqsubseteq f(d)$. Da d ein Präfixpunkt ist, gilt $f(\bot) \sqsubseteq d$. Damit folgt durch Induktion, dass $f^n(\bot) \sqsubseteq d$ und somit $\mathit{fix}(f) = \bigsqcup_{n\in\omega} f^n(\bot) \sqsubseteq d$.
Zudem ist jeder Fixpunkt ein Präfixpunkt, und es folgt, dass $\mathit{fix}(f)$ der kleinste Fixpunkt von f ist.

□

Kapitel 4

Konzepte der globalen Analyse

Das folgende Kapitel bietet einen Überblick über die Ziele, Techniken und Werkzeuge der globalen Analyse. Diese Analysen werden unter anderem im Design von Compilern eingesetzt, um globale Optimierung, wie z.B. das Entfernen von redundanten Anweisungen, zu implementieren. Die Verfahren der globalen Analyse sind aber auch in allen anderen kontrollflussbasierten Bereichen, wie z.B. der Vorhersage von Laufzeiten, anwendbar.

Allerdings sind viele der gesuchten Informationen nicht oder nur sehr schwer berechenbar. Daher gilt es, geeignete Abstraktionen zu finden, die zwar nicht das 100% richtige Ergebnis liefern, aber dennoch oftmals aussagekräftig genug sind, um aus ihnen die gewünschten Erkenntnisse zu gewinnen. Wichtig dabei ist, dass die berechneten Ergebnisse korrekt, aber nicht zwangsläufig optimal sind. Andernfalls würden Programmtransformationen, die auf diesen Ergebnissen beruhen, nicht semantikerhaltend sein und somit das Verhalten der Programme ändern. Informationen, die nicht optimal aber dennoch richtig sind, nennt man **konservativ**. Mit der Berechnung von solch konservativen Informationen beschäftigt sich das nun folgende Kapitel.

4.1 Datenflussanalyse

Eine weit verbreitete Technik, um statische Laufzeitinformationen zu erhalten, ist die sogenannte **Datenflussanalyse** ([NNH99]). Dabei berechnet man zunächst für ein gegebenes Programm den Kontrollflussgraphen[1], der alle zur Laufzeit möglichen Pfade durch ein Programm repräsentiert. Dieser Graph dient als Grundlage für die Datenflussanalyse. Dazu wird der Kontrollflussgraph mit temporären Variablen und Gleichungen beschriftet. Den so entstandenen Graphen nennt man **Datenflussgraph**.

Die einzelnen Schritte, die für eine Datenflussanalyse notwendig sind, werden im folgenden an einem Beispiel informell dargestellt. In diesem Beispiel sollen für ein gegebenes Programm die "**verfügbaren Teilausdrücke**" berechnet werden. Das Ziel

[1] Ein Kontrollflussgraph ist eine Repräsentation eines Programms, wobei Knoten Programmfragmente und Kanten mögliche Ausführungspfade darstellen.

dieser Analyse ist es, in einem Programm alle nicht-trivialen arithmetischen Teilausdrücke zu identifizieren, die mehrfach benutzt werden und deren Berechnung viel Zeit in Anspruch nimmt. Das Ergebnis der Analyse kann dann dazu verwendet werden, das Eingabeprogramm zu optimieren und schneller zu machen. Als Beispieleingabe dient das Programm aus Listing 4.1. Der dazugehörige Kontrollflussgraph ist in Abbildung 4.1 dargestellt.
Nicht-triviale arithmetische Ausdrücke sind alle arithmetischen Ausdrücke, die nicht nur aus einer Variablen oder Konstanten bestehen. An einem Programmpunkt sind alle arithmetischen Teilausdrücke verfügbar, die vorher schon einmal berechnet wurden und keine Variablen enthalten, die in dem aktuellen Programmpunkt verändert werden.

```
x = a + b;
b = x / 2 + a;
while ( a + 2 < x / 2 ){
  b = x * 2 + a;
  a = a + 3;
}
x = a / b;
```

Listing 4.1: Beispielprogramm

Um die Berechnung durchzuführen, werden alle Kanten mit zusätzlichen Variablen versehen, welche später die berechneten Datenflussinformationen enthalten werden. Die Knoten des Kontrollflussgraphen werden mit Funktionen versehen, die aus den eingehenden Werten einen neuen Datenflusswert berechnen. Im Falle der "verfügbaren Teilausdruck"-Analyse besteht ein Datenflusswert aus einer Menge von arithmetischen Ausdrücken. Da an einem Knoten nur diejenigen arithmetischen Ausdrücke verfügbar sind, die auf allen Pfaden durch ein Programm zu diesem Knoten gelangen, ist es notwendig, die Schnittmenge über alle eingehenden Informationen zu bilden und die temporären Variablen initial mit der Menge aller Teilausdrücke zu beschriften. Im Beispiel ist diese Menge:

$$AExp = \{a + b,\ x/2,\ x/2 + a,\ a + 2,\ x * 2,\ x * 2 + a,\ a + 3,\ a/b\}$$

Dies führt dann zu dem in Abbildung 4.2 dargestellten Datenflussgraphen. Der Startpunkt v_1 der Analyse wird mit der leeren Menge initialisiert. Den Datenflusswert für einen Programmpunkt erhält man nun, wenn man vom Startzustand ausgehend die in Abbildung 4.2 angegebenen Berechnungsregeln anwendet. Enthält der Kontrollflussgraph Zyklen, so muss gegebenfalls der Datenflusswert mehrfach berechnet werden - solange, bis er sich nicht mehr verändert. Das gilt für alle Programmpunkte, für die ein Pfad im Kontrollflussgraphen existiert, der einen Zyklus enthält. Das endgültige Ergebnis der Analyse ist in Tabelle 4.1 dargestellt.

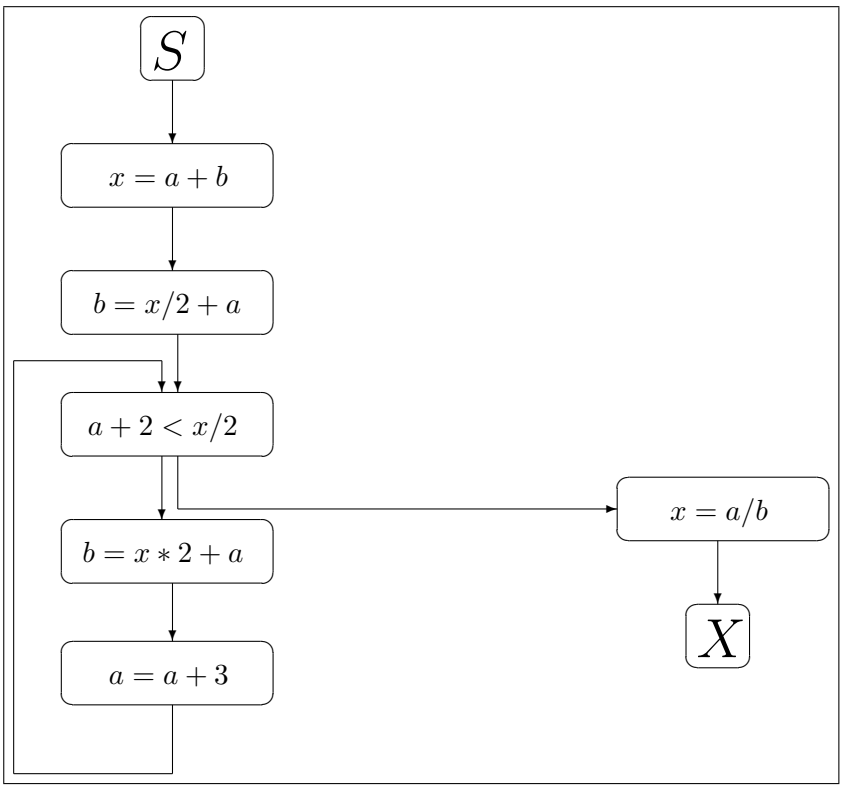


Abbildung 4.1: Kontrollflussgraph zu Listing 4.1.

Zur Lösung von Datenflussproblemen, die auf rekursiven Gleichungssystemen beruhen, existieren viele verschiedene mathematische Methoden. Die dieser Arbeit zugrundeliegende Methode ist die in Kapitel 3 beschriebene **Fixpunktiteration**. Die folgenden Abschnitte dienen zunächst dazu, die Grundlagen der Datenflussanalyse zu formalisieren. Danach werden einige iterative Algorithmen zur Lösung von Datenflussproblemen vorgestellt. Der letzte Teilabschnitt befasst sich mit Problemen und Lösungsansätzen, um **interprozeduralen Kontrollfluss** zu analysieren.

4.1.1 Grundlagen

In diesem Abschnitt werden einige grundlegende Begriffe der Datenflussanalyse eingeführt.

Definition 4.1.1 (Kontrollflussgraph (KFG)). Ein Viertupel $K = (N, E, s, x)$ mit einer Menge N von Knoten, einer Menge $E \subseteq N \times N$ von gerichteten Kanten, einem eindeutigen Startknoten s und einem eindeutigen Endknoten x heißt *Kontrollflussgraph*. Für den Startknoten s gilt:

$$\forall u \in V : (u, s) \notin E$$

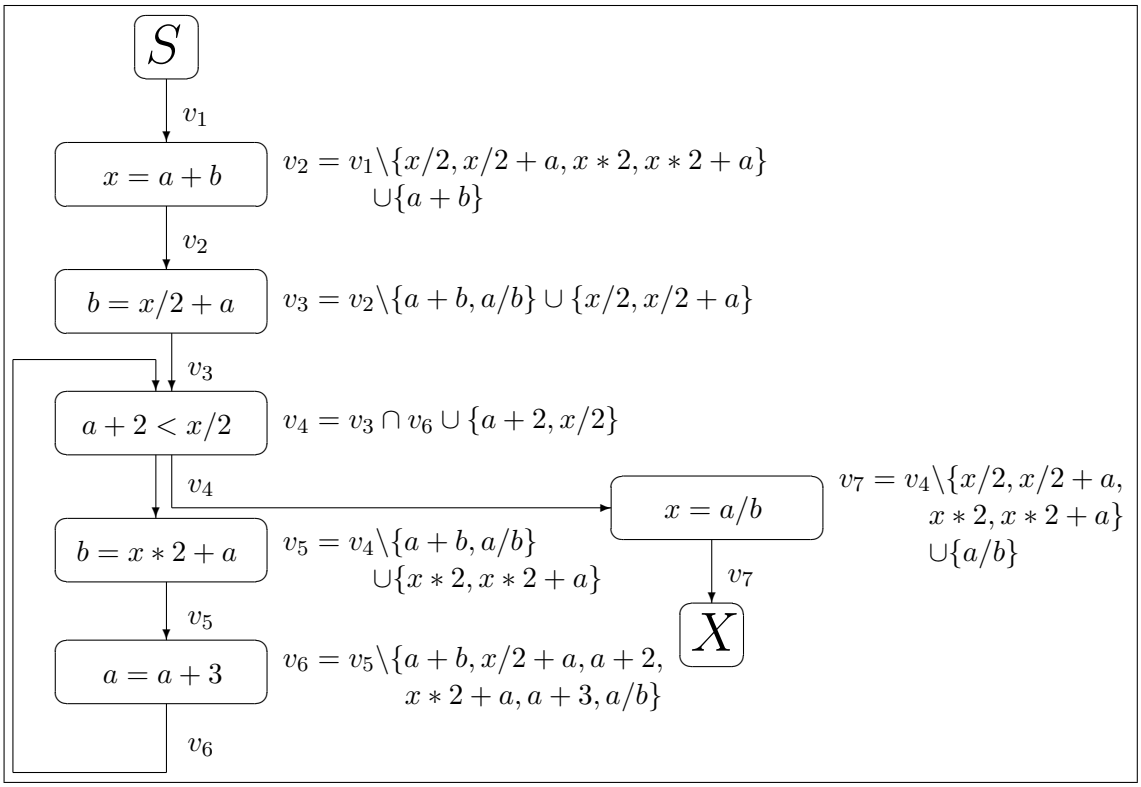


Abbildung 4.2: Datenflussgraph zu Listing 4.1.

Programmpunkt	Datenflusswert
v_1	$\{\}$
v_2	$\{a + b\}$
v_3	$\{x/2, x/2 + a\}$
v_4	$\{x/2, a + 2\}$
v_5	$\{x/2, a + 2, x * 2, x * 2 + a\}$
v_6	$\{x/2, x * 2\}$
v_7	$\{a + 2, a/b\}$

Tabelle 4.1: Ergebnisse zu Abbildung 4.2

Für den Endknoten x gilt:

$$\forall u \in V : (x, u) \notin E$$

Darüberhinaus muss es eine Funktion $F : N \to S$ geben, die den Knoten einzelne Programmfragmente zuordnet. S bezeichnet dabei die Menge der Programmfragmente, die man z.B. aus der Syntaxbaumdarstellung erhält. □

Die Forderung, dass ein Kontrollflussgraph einen eindeutigen Start- und Endknoten besitzt, ist keine Einschränkung, da jeder normale Graph einfach mit zwei zusätzlichen Knoten versehen werden kann. Die Konstruktion von Kontrollflussgraphen wird ausführlich in [All70] beschrieben.

Definition 4.1.2 (Programmpunkt). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein *Programmpunkt* ist eine Stelle im Kontrollflussgraphen K, direkt vor oder hinter einem Knoten $n \in N$. □

Definition 4.1.3 (Pfad). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein *Pfad* π in diesem Kontrollflussgraphen K ist eine Folge $n_1, n_2, \ldots, n_k$ von Knoten n_1 bis n_k für die gilt:

1. $\forall i \in \{1, ..., k\} : n_i \in N$
2. $\forall i \in \{1, ..., k-1\} : (n_i, n_{i+1}) \in E$

Der *leere Pfad* wird mit ϵ, die *Menge aller Pfade* $n_a = n_1, n_2, \ldots, n_k = n_b$ von n_a nach n_b mit $P[n_a, n_b]$ bezeichnet. □

Definition 4.1.4 (Datenflussproblem). Sei $V = (A, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband und $K = (N, E, s, x)$ ein Kontrollflussgraph. $D = (K, V, [\![\ ]\!])$ heißt *Datenflussproblem*, wenn $[\![\ ]\!] : N \to (V \to V)$ eine Funktion ist, die den Knoten des Kontrollflussgraphen Funktionen von $V \to V$ zuordnet. Sie werden als *Übergangs-* oder *Transferfunktionen* bezeichnet. □

Definition 4.1.5 (Monotones Datenflussproblem). Sei $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. D heißt *monoton*, falls für alle Knoten n des Kontrollflussgraphen K die Funktion $[\![n]\!]$ monoton ist. □

Definition 4.1.6 (Distributives Datenflussproblem). Sei $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. D heißt *distributiv*, falls für alle Knoten n von K die Funktion $[\![n]\!]$ distributiv ist. □

Definition 4.1.7 (Pfadsemantik). Sei $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. Die semantische Funktion $[\![\ ]\!]$ lässt sich folgendermaßen auf einen Pfad $\pi = n_1, n_2, ..., n_k$ erweitern:

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

□

Damit ergibt sich die Lösung eines Datenflussproblems für einen Knoten n als die Vereinigung $\sqcup$ aller Pfadsemantiken für Pfade von s nach n, angewandt auf den Startwert $\bot$.

Definition 4.1.8 (MOP-Lösung). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. Die *meet-over-all-paths*-Lösung wird für alle $n \in N$ definiert als:

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

□

Bei dieser Definition ist zu beachten, dass im Allgemeinen die Menge $P[s, n]$ der Pfade von s nach n eine unendliche Mächtigkeit hat. Daher ist diese Definition nicht als Berechnungsvorschrift zu verstehen. Es muss eine algorithmische Lösungsstrategie gesucht werden, die eine Approximation des gesuchten Ergebnisses berechnet. Diese Approximation heißt **korrekt** oder **sicher**, wenn sie größer oder gleich der gesuchten Lösung bezüglich der Ordnung $\sqsubseteq$ des gewählten Verbandes ist.
Die MOP-Lösung ist die genauste aller Approximationen und wird daher auch **präzise** genannt. Eine algorithmische Lösungsstrategie für ein Datenflussproblem ist die *maximum-fixpoint*-Lösung.

Definition 4.1.9 (MFP-Lösung). Sei $D = (K, V, [\![\ ]\!])$ mit $K = (N, E, s, x)$ ein Datenflussproblem. Die Menge der Informationen, die am Eingang in einen Knoten $n \in N$ verfügbar sind, wird mit $pre(n)$ bezeichnet, die Menge der Informationen, die am Ausgang eines Knotens gültig ist, mit $post(n)$. Dies ist auch die gesuchte $maximum - fixpoint$-Lösung. Für die Mengen $pre(n)$ und $post(n)$ gelten folgende Berechnungsregeln:

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

□

Setzt man diese beiden Gleichungen in einander ein, so ergibt sich folgende Berechnungsvorschrift für die MFP-Lösung:

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

In [KU77] wurde bewiesen, dass die MFP-Lösung korrekt für monotone Datenflussprobleme ist. Für distributive Datenflussprobleme ist die MFP-Lösung sogar gleich der MOP-Lösung.

Satz 4.1.1 (Korrektheit der MFP-Lösung). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $D = (K, V, [\![\ ]\!])$ ein monotones Datenflussproblem. Dann ist die MFP-Lösung eine *korrekte* oder *sichere* Approximation der MOP-Lösung und es gilt:

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

□

Satz 4.1.2 (Koinzidenz von MFP- und MOP-Lösung). Sei $D = (K, V, [\![\ ]\!])$ mit $K = (N, E, s, x)$ ein distributives Datenflussproblem, dann gilt:

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

□

Bei den hier vorgestellten Definitionen wurde implizit immer von einem Problem ausgegangen, bei dem an jedem Knoten die berechnete Information aus der Vereinigungsmenge der beiden Vorgängerknoten besteht. Bei dem in Abschnitt 4.1 vorgestellten Beispiel besteht die Information eines Knotens dagegen aus der Schnittmenge der Vorgängerinformationen, angewendet auf die Transferfunktion. Die obigen Definitionen sind aber dennoch ausreichend, da ein *Schnittmengenproblem* als *Vereinigungsproblem* auf dem dualen Verband angesehen werden kann.

Darüberhinaus existieren auch Probleme, die nicht durch eine *Vorwärtsanalyse* gelöst werden können. Ein bekanntes Beispiel ist das Problem der "**lebendigen Variablen**", bei dem es darum geht, ob der Wert einer Variablen im weiteren Programmablauf noch einmal benutzt wird. Die Lösung erhält man durch eine sogenannte *Rückwärtsanalyse*, bei der die Information an einem Knoten aus den Informationen der Nachfolgeknoten berechnet werden. Aber hierfür sind die obigen Definitionen ausreichend, da ein Rückwärtsproblem auf dem Graphen $K = (N, E, s, x)$ als Vorwärtsproblem mit invertiertem Kontrollflussgraphen $K^{-1} = (N, E^{-1}, x, s)$ angesehen werden kann.

4.1.2 Algorithmen

Nachdem im vorherigen Abschnitt einige grundlegende Begriffe der Datenflussanalyse eingeführt wurden, beschäftigt sich dieser Abschnitt mit grundlegenden algorithmischen Lösungsverfahren. Man unterscheidet viele verschiedene Verfahren, wobei die wichtigste Klasse, die sich über die Jahre herausgebildet hat, die Klasse der iterativen Algorithmen ist. Auf diese soll daher auch näher eingegangen werden.

```
forall n ∈ N
    MFP(n) = ⊥;
done = false;

while (!done)
    done = true;
    forall n ∈ N
        temp = [[n]](⨆{MFP(m) | (m, n) ∈ E});
        if (temp ≠ MFP(n))
            MFP(n) = temp;
            done = false;
```

Listing 4.2: Intuitive Methode zur Berechnung der MFP-Lösung

4.1.2.1 Intuitive Methode

Wie bereits erwähnt stellt die in Abschnitt 4.1.1 vorgestellte MFP-Lösung einen iterativen Ansatz dar, der als solcher implementiert werden kann. Als Eingabe erhält der Algorithmus einen Kontrollflussgraphen $K = (N, E, s, x)$, einen vollständigen Verband V und eine semantische Funktion $N \to (V \to V)$. Der Algorithmus selbst ist als Pseudo-Code in Listing 4.2 dargestellt. Das Ergebnis ist ein mit Datenflussinformationen dekorierter Kontrollflussgraph.
Die Terminierung dieses Algorithmus ist an die Bedingungen gekoppelt, dass es sich um ein monotones Datenflussproblem handelt und dass der zugrundeliegende Verband die aufsteigende Kettenbedingung erfüllt.
Die Komplexität dieser Methode wird bestimmt durch die Anzahl der Iterationen, die zur Stabilisation benötigt werden, sowie den Kosten zum Vergleich von zwei Verbandselementen.

4.1.2.2 Worklist Iteration

Die Konvergenz des intuitiven Algorithmus lässt sich beschleunigen, indem man nicht in jeder Iteration die Informationen für alle Knoten neu berechnet, sondern nur diejenigen Knoten erneut betrachtet, für die sich die Eingangsinformation geändert hat. Dazu wird eine Worklist eingeführt, die die zu betrachteten Knoten enthält. Der verbesserte Algorithmus ist in Listing 4.3 dargestellt. Dieser Algorithmus lässt sich weiter beschleunigen, indem man eine Heuristik zur Auswahl der Knoten aus der Worklist benutzt.

```
forall n ∈ N
    MFP(n) = [[n]](⊥);
workset = {s};

while (workset ≠ ∅)
    let
        n ∈ workset
    in
        workset = workset\{n};
        temp = [[n]](⨆{MFP(m) | (m,n) ∈ E});
        if (temp ≠ MFP(n))
            MFP(n) = temp;
            workset = workset ∪ {m | (n,m) ∈ E};
```

Listing 4.3: Worklist Methode zur Berechnung der MFP-Lösung

4.1.2.3 Basisblock-Optimierung

Eine andere Methode, um die Berechnungen zu beschleunigen, ist die Verwendung von Basisblöcken anstelle von normalen Knoten. Dabei fasst ein Basisblock mehrere ursprüngliche Knoten zu einem neuen Knoten zusammen.

Definition 4.1.10 (Basisblock). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Eine Folge von Knoten $(n_1, \ldots, n_k)$ heißt *Basisblock*, wenn $\forall i \in \{1, \ldots, k-1\}$ gilt:

$$\begin{aligned} & n_i \text{ ist der einzige Vorgänger von } n_{i+1} \\ \wedge \quad & n_{i+1} \text{ ist der einzige Nachfolger von } n_i \end{aligned}$$

□

Durch die Verwendung von Basisblöcken lässt sich der Iterationsprozess dadurch beschleunigen, dass nur noch für jeden dieser Blöcke der Datenflusswert gespeichert wird. Für die ursprünglichen Knoten innerhalb dieser Blöcke lässt sich der Datenflusswert nachträglich sehr schnell berechnen. Ein weiterer Vorteil dieser Methode ist der verringerte Platzbedarf, den dieser Ansatz mit sich bringt.

4.1.2.4 Widening und Narrowing

Eine Möglichkeit, um die Terminierung von Datenflussanalysen zu erzwingen, ist die Anwendung von Widening- und Narrowing-Operatoren, wie sie in [CC77] und [CC92] beschrieben werden. Damit wird die Terminierung sogar dann erreicht, wenn der zugrundeliegende Verband nicht die aufsteigende Kettenbedingung erfüllt. Aber auch

in Verbänden, die diese Eigenschaft besitzen, lässt sich die Terminierung zugunsten eines Informationsverlustes beschleunigen. Dazu definiert man einen entsprechenden *Widening*-Operator, der so gewählt sein muss, dass er ein Verbandselement durch ein größeres, welches eine sichere Approximation darstellt, abschätzt. Eine solche Kette von Abschätzungen muss so gewählt sein, dass sie endlich ist und der Prozess somit terminiert.

Die auf diese Weise gefundene Abschätzung kann man durch eine weitere Iteration unter Anwendung des *Narrowing*-Operators dem kleinsten Fixpunkt von oben wieder annähern.

Definition 4.1.11 (Widening). Sei V ein Verband und $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. Eine Abbildung $\bigtriangledown \in (V \times V \to V)$ heißt *Widening*, wenn gilt:

$$\begin{aligned} \forall a, b \in V &: a \sqsubseteq (a \bigtriangledown b) \\ \wedge\ \forall a, b \in V &: b \sqsubseteq (a \bigtriangledown b) \end{aligned}$$

Zudem soll für alle aufsteigenden Ketten $x_0 \sqsubseteq x_1 \sqsubseteq ... \sqsubseteq x_n$ die Kette $y_0, y_1, ...y_n$, definiert als

$$\begin{aligned} y_0 &= x_0 \\ y_{i+1} &= y_i \bigtriangledown x_{i+1} \end{aligned}$$

nicht streng aufsteigend sein, d.h. sich irgendwann stabilisieren. Die Iterationsfolge mit Widening für eine Funktion $F : D \to D$ ist dann definiert als

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

□

Wendet man einen Widening-Operator auf einen vollständigen Verband an, so erhält man einen Punkt, der größer oder gleich dem kleinsten Fixpunkt ist.

Unter Verwendung eines Narrowing-Operators kann man sich nun dem kleinsten Fixpunkt wiederum von oben annähern. Dazu definiert man einen Narrowing-Operator wie folgt:

Definition 4.1.12 (Narrowing). Sei V ein Verband und $D = (K, V, [\![\ ]\!])$ ein Datenflussproblem. Eine Abbildung $\bigtriangleup \in (V \times V \to V)$ heißt *Narrowing*, wenn gilt:

$$\forall a, b \in V : (a \sqsubseteq b) \Rightarrow (a \sqsubseteq (a \bigtriangleup b) \sqsubseteq b)$$

Darüberhinaus soll für alle absteigenden Ketten $x_0 \sqsupseteq x_1 \sqsupseteq ... \sqsupseteq x_n$, die Kette $y_0, y_1, ..., y_n$, definiert durch

$$\begin{aligned} y_0 &= x_0 \\ y_{i+1} &= y_i \triangle x_{i+1} \end{aligned}$$

sich irgendwann stabilisieren. Die Iterationsfolge mit Narrowing für eine Funktion $F : D \to D$ ist nun definiert als

$$\begin{aligned} X^0 &= \text{Fixpunkt der Aufwärtsiteration} \\ X^{i+1} &= \begin{cases} X^i, & \text{falls } F(X^i) \sqsupseteq X^i, \\ X^i \triangle F(X^i), & \text{sonst.} \end{cases} \end{aligned}$$

□

Betrachtet man einen vollständigen Verband, der die Kettenbedingung erfüllt, so gilt für diese, dass die Operation "kleinste obere Schranke" ein Widening, die Operation "größte untere Schranke" ein Narrowing ist.

4.1.3 Interprozeduraler Kontrollfluss

Die bisherige Diskussion über Datenflussanalyse betrachtete nur Probleme, die sich auf intraprozeduralen Kontrollfluss beschränkten. Es ist aber von größtem Interesse, Programme in ihrer ganzen Komplexität zu analysieren. Besondere Probleme bereiten dabei rekursive Funktionen und modulare Programme. Aber auch für normale Funktionsaufrufe ist ein spezielles Konzept von Nöten. Ein trivialer Ansatz um interprozeduralen Kontrollfluss zu analysieren, wäre sicherlich, alle Prozeduraufrufe so konservativ wie möglich zu behandeln. Im Falle der Datenflussanalyse bedeutet dies, anzunehmen, ein Prozeduraufruf würde alle Werte verändern. Somit müsste man alle bisherigen Datenflusswerte zerstören, was einem Verlust allen Wissens gleichkommt. Wie bereits erwähnt, ist dieses Verfahren als sehr konservativ zu bezeichnen und liefert daher auch nur sehr unbefriedigende Ergebnisse.
Eine mögliche Optimierung wäre daher, nur den Teil an Informationen zu zerstören, von dem man sicher behaupten kann, dass die aufgerufene Prozedur diese Werte verändert. Dieser Ansatz birgt aber auch nicht die Lösung des Problems. Er erfordert eine zusätzliche Analyse, um herauszufinden, welche Werte der Analyse betroffen sind.
Eine weitere Möglichkeit, interprozeduralen Kontrollfluss zu betrachten, ist die Methode des *Inlinings*. Dabei kopiert man den Prozedurrumpf an Stelle des Prozedurkopfes an die aufrufenden Stellen. Dieses Verfahren ist sehr exakt, da es alle verschiedenen Aufrufe einer Prozedur getrennt voneinander behandelt. Allerdings ist dieses Verfah-

ren nur für nicht-rekursive Prozeduren anwendbar, was es für die Praxis uninteressant macht. Außerdem kann es auch im Falle von nicht rekursiven Prozeduren zu einem exponentiellen Wachstum des Codes kommen.

Die in dieser Arbeit benutzte Methode, um interprozeduralen Kontrollfluss zu betrachten, ist der sogenannte *Call-String-Ansatz*, der in [SP81] eingeführt wurde. Dazu berechnet man für ein Programm zunächst einen einzigen zusammenhängenden Kontrollflussgraphen, indem man Prozeduraufrufe und deren Rückkehr einfach als Kontrollübergangspunkte auffasst. Nun führt man ein Tupel ein, indem man sich die Aufrufhistorie für einen Pfad durch ein Programm merkt. Dieses Tupel wird dann als *Call-String* bezeichnet. Somit lassen sich innerhalb einer Prozedur alle verschiedenen Pfade anhand dieser Aufrufhistorie unterscheiden und es kommt nicht zu einer Vermischung der Informationen durch Schrankenbildung oder ähnliches.
Nun ist es aber offensichtlich, dass diese Call-Strings beliebig lang werden können. Dies geschieht z.B. im Falle einer rekursiven Prozedur. Von daher ist es sinnvoll, die Länge eines Call-Strings zu beschränken, so dass die Länge immer kleiner oder gleich k für $k \geq 0$ ist. Dies führt auf der einen Seite dazu, dass der Call-String für einen Pfad nur endliche Länge aufweist, im Gegenzug aber auch zu einem Verlust an Informationen. Der Wert von k ist daher mit Bedacht zu wählen, er hängt aber immer von der eigentlichen Problemstellung sowie von dem zu analysierenden Programm ab.

Definition 4.1.13 (Supergraph). Sei P ein Programm, das aus den Prozeduren $P_0, P_1, ..., P_n$. Sei P_0 die Hauptprozedur des Programms P. Weiterhin seien $K_0, K_1, ..., K_n$ mit $K_i = (N_i, E_i, s_i, x_i)$ mit $i \in \{0, \ldots, n\}$ die zu $P_0, P_1, ..., P_n$ gehörendenden Kontrollflussgraphen. Dann heißt $K^* = (N^*, E^*, s^*, x^*)$ *Supergraph*, wenn gilt:

$$
\begin{aligned}
N^* &= \bigcup \{N_i \backslash \{call\ P_i \mid 0 \leq i \leq n\} \cup \{call_i, return_i \mid 0 \leq i \leq n\}\} \\
E^* &= \bigcup \{E_i \cup \{(call_i, s_i), (x_i, return_i), (call_i, return_i) \mid 0 \leq i \leq n\}\} \\
&\quad \backslash \{(u, call\ P_i), (call\ P_i, v) \mid u, v \in \bigcup_{0 \leq j \leq n} N_j, \quad 0 \leq i \leq n\} \\
&\quad \cup \{u, call_i \mid (u, call\ P_i) \in \bigcup_{0 \leq j \leq n} E_j,\ 0 \leq i \leq n\} \\
&\quad \cup \{call_i, v \mid (call\ P_i, v) \in \bigcup_{0 \leq j \leq n} E_j, 0 \leq i \leq n\}
\end{aligned}
$$

Die Knoten s^* und x^* sind die Start- bzw. Endknoten des Supergraphen mit $s^* = s_0$ und $x^* = x_0$. Jeder Prozeduraufruf $call\ P_i$ wird durch die Knoten $call_i$ und $return_i$ dargestellt. Die Transformation eines Aufrufknoten $call\ P_i$ ist in Abbildung 4.3 dargestellt. □

Die Kante zwischen $call_i$ und $return_i$ erlaubt es, Informationen, die von der aufge-

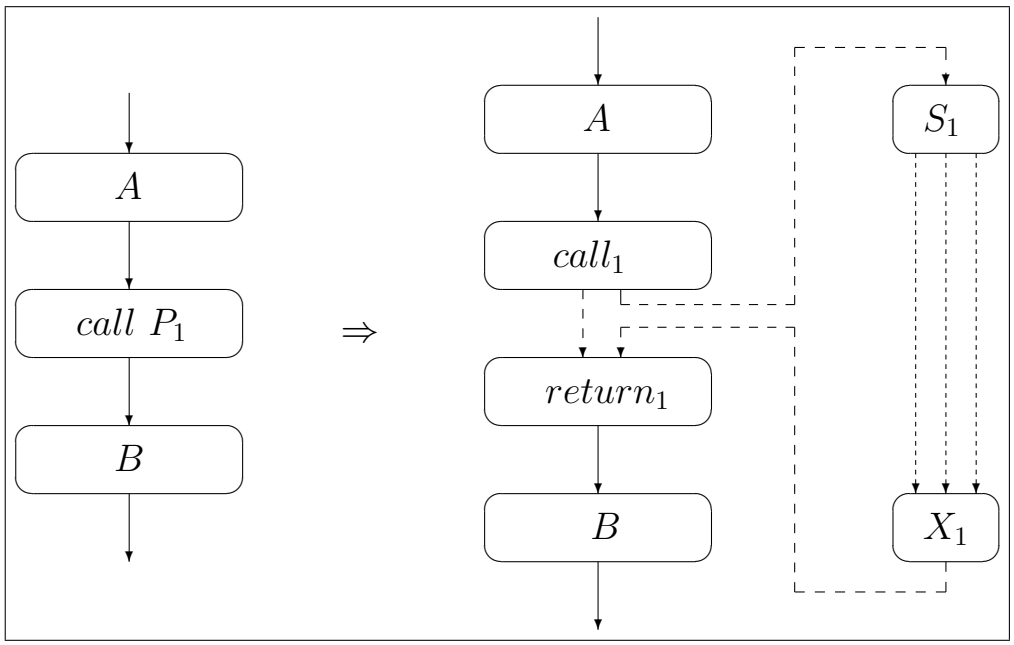


Abbildung 4.3: Transformation eines Call-Knoten im Supergraphen.

rufenen Prozedur nicht verändert werden, weiter über den Graphen zu propagieren. Probleme entstehen allerdings, wenn der Name der aufzurufenden Prozedur in einer Variable "versteckt" ist. In diesem Fall müssen Kanten zu allen möglichen Prozeduren gezogen werden, die hinter diesem Aufruf stecken können. Zusätzlich zu diesen zusätzlichen Knoten existiert für die neu eingeführten *return*-Knoten eine spezielle Funktion $return : V \times V \rightarrow V$ die anstelle der Vereinigungsfunktion für den zusammenfließenden Kontrollfluss verwendet wird.

Betrachtet man den Supergraphen wiederum als intraprozeduralen Kontrollflussgraphen, so kann man die in Abschnitt 4.1.1 beschriebene Datenflussanalyse zur Lösung von Datenflussproblemen benutzen. Den einzigen zusätzlichen Aufwand, den dieses Verfahren mit sich bringt, ist der Aufwand um den Supergraphen zu berechnen ([Mar95]).

Im folgenden werden Kontrollflussgraph und Supergraph synonym für den interprozeduralen Kontrollflussgraphen verwendet.

4.2 Abstrakte Interpretation

Abstrakte Interpretation ist ein Verfahren zur Programmanalyse. Es wurde von Cousot und Cousot im Jahr 1977 vorgestellt ([CC77] und [CC92]). Aufgrund des allgemeinen Konzeptes lassen sich mit Hilfe der Abstrakten Interpretation auch schwierige Da-

tenflussprobleme lösen. In [WM96] wurde gezeigt, dass man Datenflussanalysen als Spezialfall der Abstrakten Interpretation auffassen kann. Weitere Details dazu finden sich in [NNH99].

Ziel der Abstrakten Interpretation ist es, eine **konkrete Semantik** durch eine **abstrakte Semantik** zu ersetzen. Dabei versucht man, konkrete Werte durch abstrakte Werte so zu ersetzen, dass beide eine feste Beziehung zueinander besitzen. D.h. für jeden konkreten Wert k soll ein abstrakter Wert $\overline{k}$ existieren, der k beschreibt. Dies drückt man durch die **Abstraktionsrelation** $\lhd$ aus: $k \lhd \overline{k}$. Für jede Operation op innerhalb der konkreten Semantik muss eine solche Abstraktion $\overline{op}$ existieren und es muss gelten:

$$k_1 \lhd \overline{k_1} \wedge k_2 \lhd \overline{k_2} \Rightarrow (k_1 \ op \ k_2) \lhd (\overline{k_1} \ \overline{op} \ \overline{k_2})$$

Somit ist gewährleistet, dass die gewählte Operation $\overline{op}$ die Operation op korrekt abstrahiert. $\overline{op}$ heißt **abstrakte Operation** auf dem abstrakten Wertebereich der abstrakten Semantik, die die entsprechende Operation auf dem konkreten Wertebereich approximieren soll.

Eine große Herausforderung in der Abstrakten Interpretation ist die Wahl der abstrakten Semantik. Diese versucht man so zu wählen, dass Berechnungen auf ihr immer terminieren und die Ergebnisse Rückschlüsse auf das Verhalten des Originalprogramms zulassen.

Beispiel 4.2.1. Im nun folgenden Beispiel soll für arithmetische Ausdrücke, die nur die Operatoren $+$ und $*$ verwenden, mit Hilfe einer abstrakten Semantik das Vorzeichen bestimmt werden. Dazu wird für die abstrakte Semantik folgender Wertebereich gewählt: $\{<, 0, >, ?\}$. Das Fragezeichen steht für Werte, für die das Vorzeichen unbekannt ist. Damit ergeben sich für die abstrakten Operatoren $\overline{+}$ und $\overline{*}$ die in Tabelle 4.2 dargestellten Berechnungsregeln.

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

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

Tabelle 4.2: $\overline{+}$ und $\overline{*}$ Berechnungstabellen

□

Im folgenden soll nun Abstrakte Interpretation formal definiert werden. Da die Datenflussanalyse einen operationalen Charakter besitzt, geht sie aus einer operationalen Semantik hervor. Als Grundlage einer solchen Semantik dient eine mathematische Maschine.

Definition 4.2.1 (Mathematische Maschine). Eine *mathematische Maschine* ist ein Viertupel $M = (Z, A, E, \rightarrow)$ mit einer Menge von Zuständen Z, einer Menge von Startzuständen $A \subseteq Z$, einer Menge von Endzuständen $E \subseteq Z$ und einer Übergangsfunktion $\rightarrow \subseteq ((Z \backslash E) \times Z)$. Ein *Berechnungsschritt* ist ein Übergang von $z_i \rightarrow z_j$ mit $z_i, z_j \in Z$. Eine Folge von Berechnungsschritten $z_0 \rightarrow z_1 \rightarrow \ldots \rightarrow z_n$ heißt *Berechnung*, wenn $z_0 \in A$. Eine Berechnung *terminiert*, wenn $\nexists z_n \rightarrow z, \forall z \in Z$. Eine Berechnung heißt *fehlerfrei*, wenn $z_n \in E$, ansonsten *fehlerhaft*. Mit $\rightarrow^*$ wird die reflexive, transitive Hülle von $\rightarrow$ bezeichnet. □

Definition 4.2.2 (Abstrakte Interpretation). Seien $M = (Z, A, E, \rightarrow)$ und $\overline{M} = (\overline{Z}, \overline{A}, \overline{E}, \Rightarrow)$ mathematische Maschinen. Dann heißt $(M, \overline{M}, \delta)$ mit $\delta \subseteq (Z \times \overline{Z})$ eine *abstrakte Interpretation* von M, wenn gilt:

1. $\forall z \in A : \exists \overline{z} \in \overline{A} : z \delta \overline{z}$
2. $\forall z_i, z_j \in Z, \overline{z_i} \in \overline{Z} : z_i \delta \overline{z_i} \wedge z_i \rightarrow z_j \Rightarrow \exists \overline{z_j} \in \overline{Z} : z_j \delta \overline{z_j} \wedge \overline{z_i} \Rightarrow^* \overline{z_j}$

$\overline{M}$ simuliert die Maschine M und wird auch *abstrakte Maschine* genannt. □

In [Mar95] wurde gezeigt, wie man aus einer mathematischen Maschine und einer Problemstellung eine kumulierende abstrakte Maschine konstruieren kann. Dieser Ansatz wird in einem Werkzeug zur automatischen Generierung von Programm-Analysatoren, das im folgenden Abschnitt vorgestellt wird, realisiert.

4.3 PAG

PAG ist ein **P**rogramm-**A**nalysator-**G**enerator. Ähnlich zu den bekannten Werkzeugen flex und bison, die zur automatischen Generierung von Scannern und Parsern dienen, dient PAG dazu, Programm-Analysatoren automatisch zu generieren. Die erzeugten Analysatoren sind so effizient, dass sie nicht nur zum Testen der Analysen, sondern auch direkt für den Einsatz in der endgültigen Software geeignet sind.

Die Eingabe für PAG ist die Spezifikation einer Analyse. Daraus wird dann eine C-Bibliothek generiert. Die Analyse ist danach über eine C-Funktion aufrufbar und benötigt als Parameter nur noch den zu analysierenden Kontrollflussgraphen. Als Ergebnis erhält der Benutzer dann eine Datenstruktur, die jedem Knoten innerhalb des Kontrollflussgraphen einen Datenflusswert zuordnet. Diese Informationen können dann vom Benutzer über verschiedene Funktionen abgefragt und zum Beispiel für Optimierungen verwendet werden.

Um PAG möglichst universell einsetzbar zu machen, ist es nicht auf eine Eingabesprache beschränkt. Da alle Berechnungen auf einem Kontrollflussgraphen basieren, ist

es notwendig, einen solchen aus einem Eingabeprogramm zu generieren. Dazu gibt es in PAG sogenannte *Frontends*. Dem Benutzer ist es überlassen, beliebige neue Frontends zu implementieren, oder aus den vorhandenen ein passendes auszuwählen.

Zur Beschreibung einer Analyse stehen in PAG zwei verschiedene Sprachen zur Verfügung. Zur Beschreibung von Mengen und Verbänden dient *DATLA*. Die Beschreibung der Problemstellung, die Definition der Operatoren des Analyseverbandes sowie die Spezifikation der Übergangsfunktion erfolgt in der funktionalen Sprache *FULA*. Auf beide Sprachen wird in Kapitel 7 an geeigneten Stellen näher eingegangen. Eine Beschreibung der Konzepte von PAG findet sich in [Mar99], eine detaillierte Beschreibung der Spezifikationssprachen findet sich in [Abs02].

4.3.1 Statischer Aufrufgraph und Call-Strings

Dieser Abschnitt beschreibt kurz, wie der in Abschnitt 4.1.3 beschriebene Call-String-Ansatz in PAG realisiert wird.

Um verschiedene Aufrufpfade in einem Programm voneinander getrennt zu halten, wird jeder Knoten in PAG mit einem Feld von Datenflusswerten versehen. Einzelne Elemente können dann über ein Tupel (n, i) mit $n \in N$ angesprochen werden. Die verschiedenen Elemente beschreiben die einzelnen *Kontexte* des Knotens n.

Jede Prozedur besitzt eine feste Anzahl an Kontexten. Für jeden Knoten bezeichnet die Funktion $arr : N \rightarrow \mathbb{N}$ die Anzahl an Kontexten, die dieser Knoten besitzt.

Betrachtet man den Code aus Listing 4.4, so sieht man, dass die Prozedur $fac()$ von zwei Stellen aus aufgerufen werden kann. Abbildung 4.4 zeigt den dazugehörenden Kontrollflussgraphen. Die Kanten zwischen den einzelnen Kontexten zeigen auch gleichzeitig, wie bei einer Datenflussanalyse die Informationen zwischen den einzelnen Kontexten ausgetauscht werden. Somit ist es möglich, zwischen den beiden Aufrufen der Prozedur $fac()$ zu unterscheiden.

Weitere Details über die Umsetzung dieses Konzeptes finden sich in [Abs02].

```
int fac ( int n )
{
  if ( n < 2 )
    return 1;
  else
    return n * fac( n - 1 );
}

void main ( int argc, char** argv )
{
  int res;
```

```
12    int n;
      scanf( "Zahl: %d", &n );
14    res = fac( n );
      printf( "Ergebnis: %d\n", res );
16  }
```

Listing 4.4: Beispielprogramm: rekursive Fakultätsfunktion

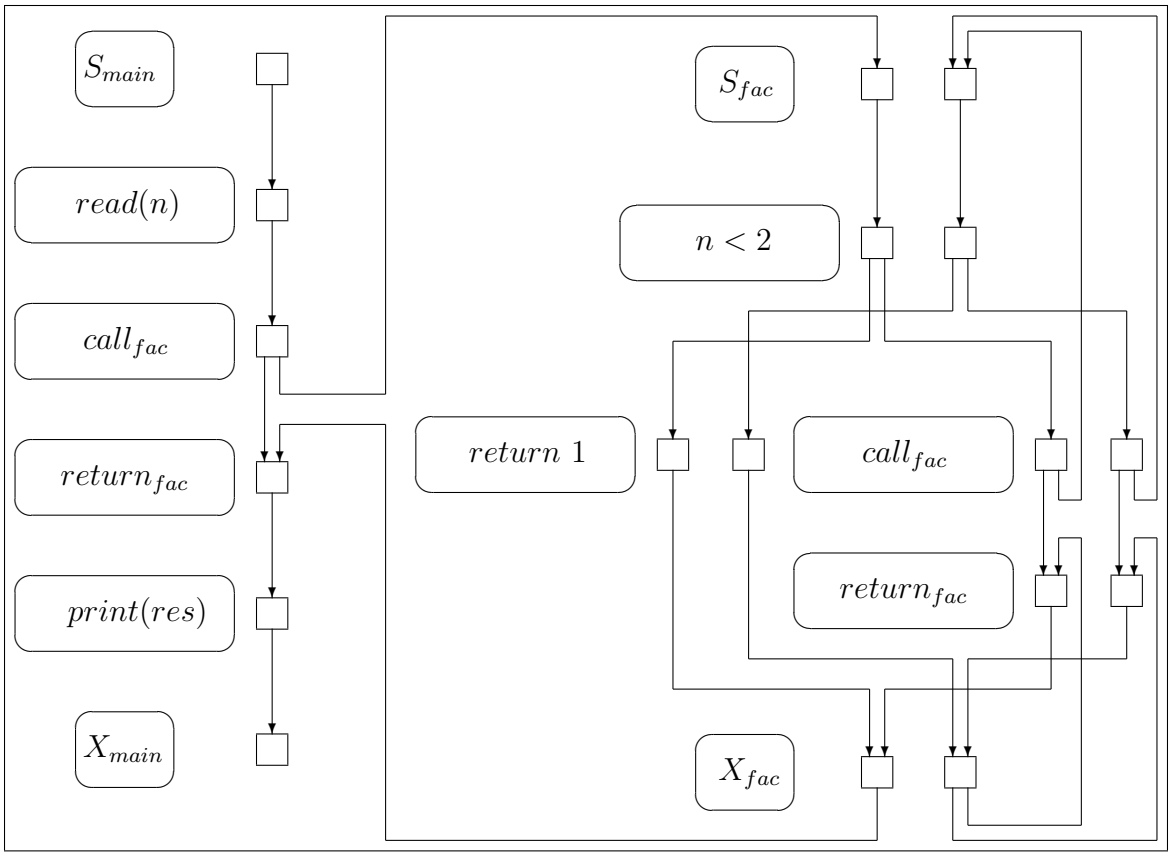


Abbildung 4.4: Kontrollabhängigkeitsgraph zu Listing 4.4.

Kapitel 5

Statisches Rückwärts-Slicing

Das folgende Kapitel beschäftigt sich mit dem Thema des statischen Rückwärts-Slicings. Ziel dieser Arbeit ist es, Programm-Slices auf Disassemblerebene durch abstrakte Interpretation zu berechnen. Wie bereits in Kapitel 2 beschrieben, existieren verschiedene Methoden, einen Slice zu berechnen. Eine davon ist die datenflussbasierte Methode, die, wie in Kapitel 4 bereits erwähnt, als Spezialform der abstrakten Interpretation aufgefasst werden kann.

Im folgenden sollen zunächst die Konzepte des datenflussbasierten Slicings näher vorgestellt werden. Anschließend wird eine abstrakte Semantik vorgestellt, die als Basis für das Slicing dienen soll. Zur Erzeugung sowie zur Analyse dieser Semantik stehen verschiedene Werkzeuge zur Verfügung. Darauf aufbauend werden verschiedene Analysen vorgestellt, mit deren Hilfe man Abhängigkeiten von Programmpunkten untereinander berechnen kann. Die Ergebnisse erlauben dann die Berechnung von Slices für beliebige Kriterien. Im letzten Abschnitt werden dann noch einige Optimierungen vorgestellt.

5.1 Datenflussbasiertes Slicing

Im Jahr 1979 stellte Weiser einen iterativen Lösungsansatz zur Berechnung von Slices vor ([Wei79], [Wei82] und [Wei84]). Er berechnete ein ausführbares Programm aus einem Originalprogramm, indem er 0 oder mehr Anweisungen löschte. Dieses berechnete Teilprogramm musste gewisse Anforderungen erfüllen: Für Variablen, die zu einem Slice gehören, müssen während der Ausführung die gleichen Werte berechnet werden wie im Originalprogramm. Die Forderung, dass ein Slice ein ausführbares Programm sein muss, wurde im folgenden wieder verworfen. Die Basisidee, die dieser Arbeit zu Grunde liegt, ist aber immer noch mit den von Weiser vorgestellten *Datenflussgleichungen* eng verwandt, lediglich eine Erweiterung auf interprozeduralen Kontrollfluss wurde vorgenommen. Im folgenden sollen zunächst einige wesentliche Begriffe des datenflussbasierten Slicings eingeführt werden.

Definition 5.1.1. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $F : N \to S$ eine Funktion, die jedem Knoten das dazugehörige Programmfragment zuordnet. Für alle $n \in N$ bezeichnet die Menge $var(n)$ alle Variablen, die in dem Programmfragment $F(n)$ vorkommen. □

Definition 5.1.2 (Slicing-Kriterium). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $F : N \to S$ die Funktion, die jedem Knoten das dazugehörende Programmfragment zuordnet. Ein Tupel $C = (n, V)$ mit $n \in N$ und $V \subseteq var(n)$ heißt *Slicing-Kriterium*. □

Definition 5.1.3 (Slice). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph, P ein Eingabeprogramm und $F : N \to S$ die Funktion, die jedem Knoten das dazugehörende Programmfragment zuordnet. Ein *Slice* hinsichtlich eines Slicing-Kriteriums $C = (n, V)$ ist eine Teilmenge $S' \subseteq N$ der Knoten von K, die folgende Bedingung erfüllt:
Sei P' das Programm, das durch S' beschrieben wird. Wenn P für eine Eingabe hält, dann muss auch P' für diese Eingabe anhalten und für alle $v \in V$ muss gelten, dass P' die gleichen Werte für v berechnet, wie P, wenn das Programmfragment $F(n)$ mit $n \in S'$ ausgeführt wird. □

Es ist offensichtlich, dass zu jedem Eingabeprogramm P und jedem Slicing-Kriterium (n, V) mindestens ein Slice S' existiert, nämlich P selbst. Dies stellt aber in den meisten Fällen eine sehr konservative Lösung dar, die meist unbefriedigend ist. Man sucht daher eine Teilmenge $S' \subseteq S$, die möglichst klein ist.

Definition 5.1.4 (Minimaler Slice). Ein Slice S' bezüglich eines Slicing-Kriteriums $C = (n, V)$ heißt *minimal*, wenn kein anderer Slice S'' hinsichtlich C existiert, der weniger Anweisungen enthält. □

Nach Weiser sind minimale Slices nicht unbedingt eindeutig und das Problem der Bestimmung dieser minimalen Slices ist nicht entscheidbar. Daher muss ein Weg gefunden werden, Approximationen an diese minimalen Slices zu berechnen. Die folgenden Definitionen dienen dazu, einen iterativen Ansatz zu beschreiben, mit dem solche Approximationen berechnet werden können.

Definition 5.1.5. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $F : N \to S$ eine Funktion, die jedem Knoten das dazugehörige Programmfragment zuordnet. Für alle $n \in N$ bezeichnet die Menge $def(n) \subseteq var(n)$ alle Variablen, deren Werte in dem Programmfragment $F(n)$ verändert werden. □

Definition 5.1.6. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $F : N \to S$ eine Funktion, die jedem Knoten das dazugehörige Programmfragment zuordnet. Für alle $n \in N$ bezeichnet die Menge $use(n) \subseteq var(n)$ alle Variablen, deren Werte in dem Programmfragment $F(n)$ benutzt werden. □

Damit lässt sich die Abhängigkeit zwischen zwei Knoten wie folgt definieren:

Definition 5.1.7 (Datenabhängigkeit). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $F : N \to S$ die Zuordnungsfunktion. Ein Knoten $j \in N$ ist von einem Knoten $i \in N$ *datenabhängig*[1], wenn es eine Variable $r \in var(j)$ gibt, für die gilt:

$$\begin{aligned} & r \in def(i), \\ \wedge \quad & r \in use(j), \\ \wedge \quad & \exists \pi \in P[i,j] : \forall n \in \pi \backslash \{i\} : r \notin def(n) \end{aligned}$$

Dies drückt man durch die Relation $\Downarrow_r$ aus: $i \Downarrow_r j$.
Die Menge aller Knoten, an denen eine Variable r definiert wurde, ist für einen Knoten n definiert als:

$$data(n, r) = \{u | u \in N \wedge u \Downarrow_r n\}$$

□

Beispiel 5.1.1. Abbildung 5.1 zeigt den zu Listing 4.1 dazugehörigen Datenabhängigkeitsgraphen. Dabei verbinden durchgezogene Kanten jeweils die Definition mit der Benutzung einer Variable. Die gestrichelten Linien zeigen den Kontrollfluss. □

Hängt die Ausführung einer Instruktion von der Ausführung einer anderen Instruktion ab, so spricht man von **Kontrollabhängigkeit**. Zur formalen Definition benötigt man zusätzlich den Begriff der **Post-Dominanz**, der wie folgt definiert ist:

Definition 5.1.8 (Post-Dominanz). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein Knoten $i \in N$ wird von einem Knoten $j \in N$ *post-dominiert*, wenn gilt:

$$\forall \pi \in P[i, x] : j \in \pi$$

Dies drückt man dann durch die Relation $\uparrow$ aus: $j \uparrow i$.
Die Menge der Post-Dominatoren für einen Knoten $n \in N$ ist definiert als:

$$pdom(n) = \bigcup \{u | u \in N \wedge u \uparrow n\}$$

□

Mit Hilfe dieser Definition lässt sich die Kontrollabhängigkeit wie folgt definieren:

Definition 5.1.9 (Kontrollabhängigkeit). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein Knoten $j \in N$ ist von einem Knoten $i \in N$ *kontrollabhängig*, wenn gilt:

$$\begin{aligned} & \exists \pi \in P[i,j] : \forall n \in \pi \backslash \{i,j\} : j \in pdom(n) \\ \wedge \quad & j \notin pdom(i) \end{aligned}$$

[1] In dieser Arbeit wird nur die Flussabhängigkeit als Datenabhängigkeit betrachtet.

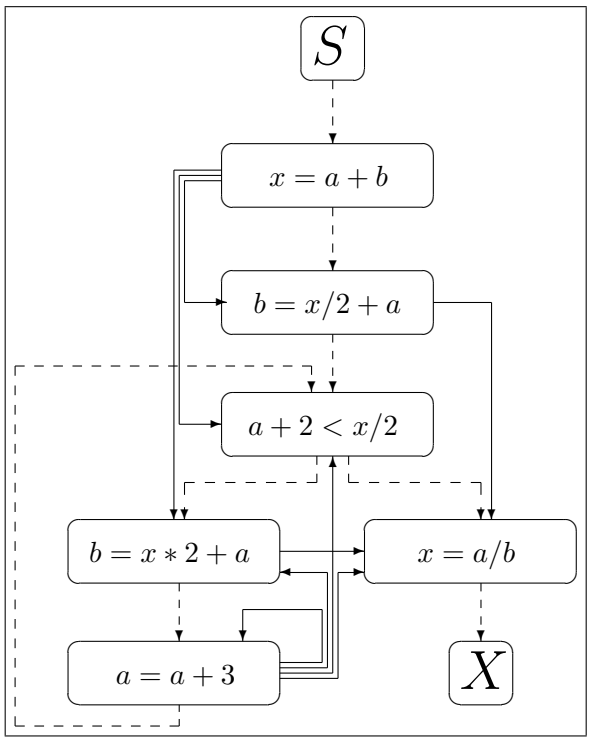


Abbildung 5.1: Datenabhängigkeitsgraph zu Listing 4.1.

Dies drückt man durch die Relation $\downarrow$ aus: $i \downarrow j$.
Die Menge der kontrollabhängigen Knoten von einem Knoten n ist definiert als:

$$\mathit{infl}(n) = \{u \mid u \in N \wedge n \downarrow u\}$$

$\mathit{infl}(n)$ heißt *Einflussbereich* von Knoten n. □

Beispiel 5.1.2. Abbildung 5.2 zeigt den Kontrollabhängigkeitsgraphen zu Listing 4.1. Die gestrichelten Kanten zeigen den Kontrollfluss, die durchgezogenen Kanten zeigen die Kontrollabhängigkeit. □

Mit Hilfe der Definitionen 5.1.7 und 5.1.9 lässt sich nun eine Approximation an minimale Slices durch einen iterativen Prozess berechnen, bei dem man aufeinanderfolgende Mengen von **relevanten Variablen** für jeden Knoten des Supergraphen berechnet.

Definition 5.1.10 (direkt relevante Variablen). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $C = (n, V)$ mit $n \in N$ und $V \subseteq \mathit{var}(n)$ ein Slicing-Kriterium. Die

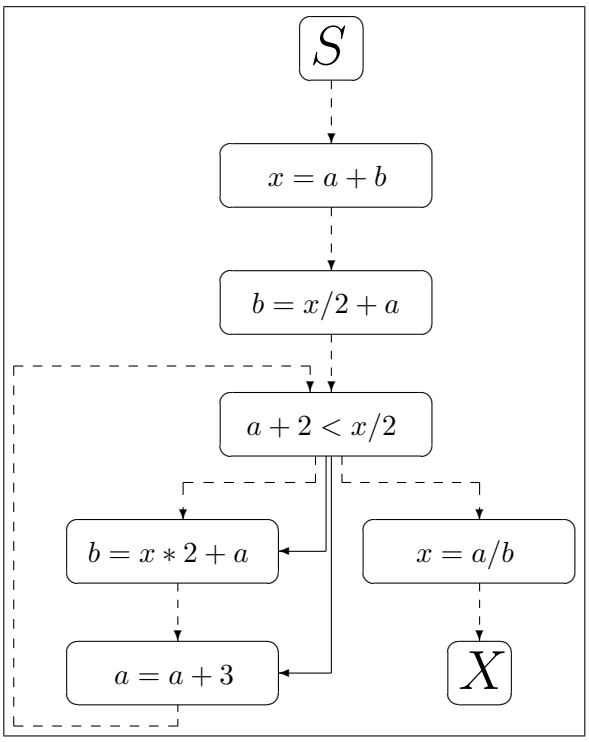


Abbildung 5.2: Kontrollabhängigkeitsgraph zu Listing 4.1.

Menge der *direkt relevanten Variablen* R_C^0 für einen Knoten $i \in N$ ist definiert als:

$$R_C^0(i) = \begin{cases} V, & \text{wenn } i = n, \\ \{v \mid \forall (i,j) \in E : (v \in R_C^0(j) \wedge v \notin def(i)) & \\ \quad \vee (v \in use(i) \wedge def(i) \cap R_C^0(j) \neq \emptyset)\}, & \text{sonst.} \end{cases}$$

□

Mit Hilfe dieser Definition kann man dann die Menge der direkt relevanten Anweisungen definieren:

Definition 5.1.11 (direkt relevante Anweisungen). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $C = (n, V)$ mit $n \in N$ und $V \subseteq var(n)$ ein Slicing-Kriterium. Die Menge der *direkt relevanten Anweisungen* S_C^0 ist definiert als die Menge aller Knoten $i \in N$, die eine Variable v definieren, die für einen Nachfolger von i im Supergraphen relevant ist.

$$S_C^0 = \{i \mid i \in N, (i,j) \in E : def(i) \cap R_C^0(j) \neq \emptyset\}$$

□

Aufbauend auf diesen Definitionen lassen sich nun ausgehend von R_C^0 und S_C^0 induktiv die Mengen R_C^k, S_C^k und B_C^k für $k \geq 0$ definieren.

Alle Variablen, die in einer Kontrollanweisung referenziert werden, sind indirekt für die Berechnung des Slices relevant, wenn ein von dieser Anweisung kontrollabhängiger Knoten relevant ist. Betrachtet man ein Slicing-Kriterium, so ist dieses zusätzlich von allen Knoten abhängig, die eine Kontrollabhängigkeit auf den betrachteten Punkt haben. Allgemein muss man diese Kontrollabhängigkeit für alle relevanten Programmpunkte betrachten und kann somit die Menge der relevanten Kontrollanweisungen definieren als:

Definition 5.1.12 (relevante Kontrollanweisungen). Die Menge der *relevanten Kontrollanweisungen* B_C^k ist definiert durch den Einfluss, den sie auf die Menge aller relevanten Anweisungen S_C^k hat:

$$B_C^k = \{b \mid i \in S_C^k, i \in \mathit{infl}(b)\}$$

□

Definition 5.1.13 (indirekt relevante Variablen). Die Menge der *indirekt relevanten Variablen* R_C^{k+1} ist definiert als die Menge aller relevanten Variablen $R_C^k(i)$ vereinigt mit der Menge aller direkt relevanten Variablen bezüglich der Kontrollanweisungen B_C^k:

$$R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b,use(b))}(i)$$

□

Definition 5.1.14 (indirekt relevante Anweisungen). Die Menge der *indirekt relevanten Anweisungen* S_C^{k+1} besteht aus allen Kontrollanweisungen B_C^k sowie allen Knoten i, die eine Variable definieren, die Relevanz für einen Nachfolger von i hat:

$$S_C^{k+1} = B_C^k \cup \{i \mid (i,j) \in E^*, \mathit{def}(i) \cap R_C^{k+1}(j) \neq \emptyset\}$$

□

Die beiden Folgen $\left(R_C^k\right)_{k \in I\!N}$ und $\left(S_C^k\right)_{k \in I\!N}$ stellen nicht abnehmende Teilmengen von Programmpunkten dar. Berechnet man den Fixpunkt der Folge $\left(S_C^{k+1}\right)_{k \in I\!N}$, so erhält man einen Slice hinsichtlich des Slicing-Kriteriums.

Satz 5.1.1. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $C = (n, V)$ mit $n \in N^*$ und $V \subseteq var(n)$ ein Slicing-Kriterium. Der Fixpunkt der Folge $\left(S_C^{k+1}\right)_{k \in I\!N}$ stellt einen sicheren Slice bezüglich des Slicing-Kriteriums C dar. □

Die Definition von S_C^{k+1} stellt einen iterativen Ansatz zur Verfügung, mit dem man Approximationen an einen minimalen Slice berechnen kann. Die folgenden Abschnitte

beschreiben nun zunächst näher den Rahmen dieser Arbeit und zeigen dann Analysen, mit deren Hilfe man die Abhängigkeiten in einem interprozeduralem Kontrollflussgraphen berechnen kann. Darauf aufbauend wird dann der Slicing-Algorithmus beschrieben.

5.2 Ausgangspunkt

Ziel dieser Arbeit ist es, einen generischen Algorithmus für das statische Rückwärts-Slicing von disassemblierten Code zu entwickeln. Es stehen Werkzeuge zum Disassemblieren von ausführbaren Programmen und zur Überführung in eine Zwischendarstellung zur Verfügung. Diese Zwischendarstellung heißt CRL (**C**ontrol Flow **R**epresentation **L**anguage) und dient der Beschreibung des Kontrollflusses eines Programms. Eine nähere Beschreibung von CRL folgt im nächsten Abschnitt. Eine CRL-Darstellung kann mit Hilfe des CRL-Frontends von PAG eingelesen und in eine den Kontrollflussgraphen beschreibende Datenstruktur übersetzt werden.

Auf dieser Basis aufbauend müssen für einen Kontrollflussgraphen mit Hilfe von verschiedenen, im folgenden noch vorgestellten Analysen, die Daten- und Kontrollabhängigkeiten berechnet werden. Danach können für beliebige Kriterien Slices berechnet werden. Das nun folgende Kapitel beschreibt nun zunächst näher die Zwischendarstellung CRL, die als abstrakte Syntax der Analysen dienen soll.

5.3 CRL-Beschreibung

CRL steht für **C**ontrol Flow **R**epresentation **L**anguage. Diese Sprache wurde im Rahmen des *Transferbereichs 14* entwickelt und beschreibt in textueller Form den Kontrollflussgraphen eines Programms. Die Sprache wurde speziell für Analysen und Optimierungen entwickelt. Die zugrundeliegende Struktur ist hierarchisch organisiert: ein Graph besteht dabei aus Instruktionen, Basisblöcken und Routinen, wobei das erstere immer im letzteren enthalten ist.

Definition 5.3.1 (Wert eines Attributs). Sei $K = (N, E, s, x)$ der aus einer CRL-Datei entstandene Kontrollflussgraph. Der *Wert eines Attributs attr* an einer Instruktion $n \in N$ kann mit $attr(n)$ abgefragt werden. □

Beispiel 5.3.1. Listing 5.1 zeigt eine Beispiel CRL-Datei. Der dazugehörende Kontrollflussgraph ist in Abbildung 5.3 dargestellt. Des weiteren zeigt dieses Beispiel die Verwendung der CRL-Attribute. Der Wert des Attributs dst_2 an Instruktion $0x0$ ist z.B. "$r0$". □

```
start with main;

routine main: name="main"
{
  entry b0: id="b0"
  {
    edges to b1/f, b2/t;
    contains {
      0x00000000:4 "mov r0, 0": src1="al", dst2="r0", src3="0",
          guard="always";
      0x00000004:4 "mov r1, 1": src1="al", dst2="r1", src3="1",
          guard="always";
      0x00000008:4 "cmp r1, r0": src1="al", src2="r1", src3="r0",
          dst4="CPSR", guard="always";
      0x0000000C:4 "bgt b2": src1="gt", src2="0x14", dst3="PC",
          guard="conditional";
    }
  }

  block b1: id="b1"
  {
    edges to x;
    contains {
      0x00000010:4 "mov r0, 0": src1="al", dst2="r0", src3="0",
          guard="always";
    }
  }

  block b2: id="b2"
  {
    edges to x;
    contains {
      0x00000014:4 "mov r0, 1": src1="al", dst2="r0", src3="1",
          guard="always";
    }
  }

  exit x;
}

end
```

Listing 5.1: CRL-Beschreibung eines Kontrollflussgraphen

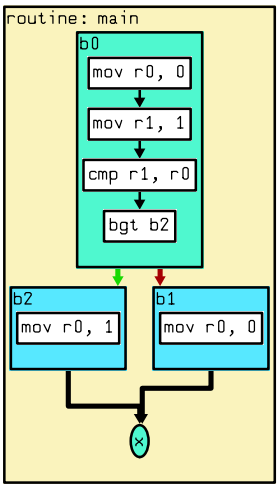


Abbildung 5.3: Kontrollflussgraph zu Listing 5.1

Die durch die zur Verfügung stehenden Werkzeuge erzeugten CRL-Dateien sind bereits für alle Instruktionen bezüglich der $def()$ und $use()$ Eigenschaft klassifiziert. D.h. die Attribute einer Instruktion geben Auskunft darüber, welche der verwendeten Ressourcen benutzt und welche verändert werden. Diese Klassifizierung stellt eine Obermenge der tatsächlich von der Instruktion veränderten Ressourcen dar. Weiterhin steht noch ein Attribut zur Verfügung, das Aussagen darüber zulässt, ob eine Instruktion immer ($guard = always$), unter bestimmten Vorbedingungen ($guard = conditional$) oder nie ($guard = never$) ausgeführt wird. Diese Klassifizierung ist für Prozessor-Architekturen interessant, die eine Form von "guarded execution[2]" umsetzen. Zudem ist dieses Attribut für die im folgenden Abschnitt beschriebene Datenabhängigkeitsanalyse notwendig.

Definition 5.3.2 (Ziele einer Instruktion). Gegeben eine CRL-Datei. Sei $K = (N, E, s, x)$ der dazugehörende Kontrollflussgraph und $Resource$ die Menge aller Ressourcen für eine Prozessor-Architektur. Die *Ziele einer Instruktion* $n \in N$ sind definiert als:

$$dst(n) = \left(\bigcup_{i \in I\!N} dst_i(n) \right) \cap Resource$$

□

Durch den Schnitt der $dst_i()$-Attribute mit der Menge aller Ressourcen einer Prozessor-Architektur werden alle Pseudo-Ziele direkt aus der Menge der Ziele ausgeschlossen.

[2]Ein Guard ist bildlich gesprochen ein Wächter, der die Ausführung einer Instruktion an gewisse Vorbedingungen knüpft.

Definition 5.3.3 (Quellen einer Instruktion). Gegeben eine CRL-Datei. Sei $K = (N, E, s, x)$ der dazugehörende Kontrollflussgraph und *Resource* die Menge aller Ressourcen für eine Prozessor-Architektur. Die *Quellen einer Instruktion* $n \in N$ sind definiert als:

$$src(n) = \left(\bigcup_{i \in \mathbb{N}} src_i(n) \right) \cap Resource$$

□

Definition 5.3.4 (Ressourcen einer Instruktion). Gegeben eine CRL-Datei und $K = (N, E, s, x)$ der dazugehörende Kontrollflussgraph. Die *Ressourcen einer Instruktion* $n \in N$ sind definiert als:

$$res(n) = dst(n) \cup src(n)$$

□

Im Idealfall beschreibt die durch die externen Werkzeuge entstandene CRL-Datei den Kontrollflussgraphen des Eingabeprogramms exakt, ansonsten wird eine sichere Annäherung an den idealen Kontrollflussgraphen berechnet. D.h. kann für einen Aufruf die entsprechende Prozedur nicht exakt vorbestimmt werden, werden Kanten vom Aufrufer zu allen möglichen aufgerufenen Prozeduren gezogen. Dies ist in [Wil01] und [KW02] auf Ebene von Assemblerprogrammen beschrieben. Auf dem so erzeugten Kontrollflussgraphen aufbauend müssen nun zunächst die Abhängigkeiten der einzelnen Instruktionen, bzw. Knoten voneinander berechnet werden.

5.4 Datenabhängigkeitsanalyse

Um einen Slice zu berechnen, ist es zunächst notwendig, die Abhängigkeiten der Ressourcen voneinander zu kennen. Diese Datenabhängigkeiten kann man mit der "**verfügbaren Definitionen**" Analyse berechnen. Dabei ist die Definition einer Ressource in Instruktion i an einer Instruktion j verfügbar, wenn es mindestens einen Pfad im Kontrollflussgraphen gibt, der keine neue Definition der betrachteten Ressource beinhaltet.

Definition 5.4.1 (verfügbare Definitionen). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Eine Ressource $r \in Resource$, die in einem Knoten $i \in N$ verändert wurde, ist an einem Knoten $j \in N$ *verfügbar*, wenn gilt:

$$\begin{aligned} & r \in dst(i) \\ \wedge \quad & \exists \pi \in P[i, j] : \forall n \in \pi \backslash \{i\} : r \notin dst(n) \end{aligned}$$

□

Wie man leicht sieht, ist das Ergebnis einer Analyse auf Basis dieser Definition eine Obermenge der Datenabhängigkeit, wie sie in 5.1.7 definiert wurde. Für den Wertebereich der benötigten Analyse muss folgendes gelten: Es ist für jeden Programmpunkt notwendig, jeder Ressource einer Prozessorarchitektur eine Menge von Knoten zuzuordnen, an denen diese Ressource zuletzt definiert wurde. Man benötigt also eine Abbildung

$$map \equiv \{f \mid f : Resource \rightarrow \mathcal{P}(N)\}$$

für die folgende Ordnung $\sqsubseteq_{map}$ für $f, g \in map$ gilt:

$$\forall x \in Resource : f(x) \subseteq g(x) \Rightarrow f \sqsubseteq_{map} g$$

Die Operatoren $\sqcup_{map}$ und $\sqcap_{map}$ sind wie folgt für $f, g \in map$ definiert:

$$\begin{aligned} f \sqcup_{map} g &\equiv \lambda x \in Resource : f(x) \cup g(x) \\ f \sqcap_{map} g &\equiv \lambda x \in Resource : f(x) \cap g(x) \end{aligned}$$

Um später beschriebene Optimierungen einfach umsetzen zu können muss der Wertebereich noch mit einem zusätzlichen größten und kleinsten Element $\top$ und $\bot$ erweitert werden. Somit ergibt sich der Wertebereich für die Datenflussanalyse als

$$func \equiv map \cup \{\bot, \top\}$$

mit der folgenden Ordnung $\sqsubseteq_{func}$:

$$\begin{aligned} & \forall y \in func : && \bot \sqsubseteq_{func} y \sqsubseteq_{func} \top \\ \wedge \quad & x_1, x_2 \in map : && x_1 \sqsubseteq_{map} x_2 \Rightarrow x_1 \sqsubseteq_{func} x_2 \end{aligned}$$

Die Operatoren $\sqcup_{func}$ und $\sqcap_{func}$ für $x, y \in func$ sind wie folgt definiert:

$$x \sqcup_{func} y \equiv \begin{cases} \top, & \text{wenn } x = \top \vee y = \top, \\ x, & \text{wenn } y = \bot, \\ y, & \text{wenn } x = \bot, \\ x \sqcup_{map} y, & \text{sonst.} \end{cases}$$

$$x \sqcap_{func} y \equiv \begin{cases} \bot, & \text{wenn } x = \bot \vee y = \bot, \\ x, & \text{wenn } y = \top, \\ y, & \text{wenn } x = \top, \\ x \sqcap_{map} y, & \text{sonst.} \end{cases}$$

Für die Berechnung der Datenflusswerte führt man nun eine Vorwärtsanalyse auf den Knoten des Kontrollflussgraphen durch und bildet an Knoten mit mehreren

Vorgängern die Vereinigung $\sqcup_{func}$ über die eingehenden Informationen. Der Verband für diese Datenflussanalyse sieht also folgendermaßen aus:

$$V_{rd} = (func, \bot, \top, \sqsubseteq_{func}, \sqcup_{func}, \sqcap_{func})$$

Für die Berechnung der Datenflusswerte spielt zusätzlich das oben angesprochene *guard*-Attribut eine wichtige Rolle. Wie in Abschnitt 5.3 bereits erwähnt, kann dieses Attribut drei verschiedene Werte annehmen: *always*, *conditional* oder *never*. Die Bedeutung wurde bereits in Abschnitt 5.3 vorgestellt. Während eine Instruktion, deren *guard*-Attribut *never* ist, in dieser Analyse komplett ignoriert werden kann, da sie nie ausgeführt wird, bedeutet *always* das genaue Gegenteil. Interessant ist besonders der Wert *conditional*, der aussagt, dass eine Instruktion eventuell ausgeführt wird.

Nun kann man zwei Funktionen $must : V_{rd} \times Resource \times N \to V_{rd}$ und $may : V_{rd} \times Resource \times N \to V_{rd}$ wie folgt definieren:

Definition 5.4.2 (*must*-Update). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein *must-Update* $must : V_{rd} \times Resource \times N \to V_{rd}$ für eine Ressource $r \in Resource$ und einen Knoten $n \in N$ für ein Verbandselement $v \in V_{rd}$ ist definiert als:

$$v(r) = \{n\}$$

□

Definition 5.4.3 (*may*-Update). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein *may-Update* $may : V_{rd} \times Resource \times N \to V_{rd}$ für eine Ressource $r \in Resource$ und einen Knoten $n \in N$ für ein Verbandselement $v \in V_{rd}$ ist definiert als:

$$v(r) = v(r) \cup \{n\}$$

□

Definition 5.4.4 (Transfer-Funktion). Sei $K = (N, E, s, x)$ der Kontrollflussgraph einer CRL-Datei. Die *Transfer-Funktion* der Datenabhängigkeitsanalyse $transfer_{rd} : V_{rd} \to V_{rd}$ für einen Knoten $n \in N$ ist definiert als:

$$\forall r \in dst(n): \quad v(r) = \begin{cases} must(v, r, n), & \text{wenn } guard(n) = always, \\ may(v, r, n), & \text{wenn } guard(n) = conditional, \\ v(r), & \text{wenn } guard(n) = never. \end{cases}$$
$$\wedge \quad \forall r \notin dst(n): \quad v(r) = v(r)$$

□

Für die Return-Funktion $return_{rd} : V_{rd} \times V_{rd} \to V_{rd}$ gilt:

$$return_{rd} = \sqcup_{func}$$

Nun definiert man die Funktion $[\![\ ]\!]_{rd} : N \to (V_{rd} \to V_{rd})$ als

$$[\![n]\!]_{rd} = transfer_{rd}$$

und erhält das Datenflussproblem $D_{rd} = (K, V_{rd}, [\![\ ]\!]_{rd})$, das monoton ist.

Um die Ergebnisse dieses Datenflussproblems abzufragen, dient die Funktion $Datadep()$, die wie folgt definiert ist:

Definition 5.4.5 (Datenabhängigkeitsfunktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph, D_{rd} das oben beschriebene Datenflussproblem und $dfi_{rd} : N \to V_{rd}$ die Funktion, die zu einem Knoten die Lösung des Datenflussproblems D_{rd} zurückliefert. Die *Datenabhängigkeitsfunktion* $Datadep : N \times Resource \to \mathcal{P}(N)$ für einen Knoten $n \in N$ und eine Ressource $r \in Resource$ ist definiert als:

$$Datadep(n, r) = dfi_{rd}(n)(r)$$

□

Satz 5.4.1 (Korrektheit). Gegeben eine CRL-Datei. Sei $K = (N, E, s, x)$ der dazugehörende Kontrollflussgraph. Sei $D_{rd} = (K, V_{rd}, [\![\ ]\!]_{rd})$ das oben beschriebene Datenflussproblem. Die MFP-Lösung von D_{rd} ist eine sichere Approximation der MOP-Lösung und es gilt:

$$\forall n \in N : \forall r \in Resource : m \in data(n, r) \Rightarrow m \in Datadep(n, r)$$

□

Beweis 5.4.1. Um die Korrektheit zu zeigen, muss folgendes gezeigt werden:

1. Das Datenflussproblem D_{rd} ist monoton $\Rightarrow$ MFP-Lösung sicher.
2. Die Aussage $\forall n \in N : \forall r \in Resource \Rightarrow m \in data(n, r) : m \in Datadep(n, r)$ ist korrekt.

Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $D_{rd} = (K, V_{rd}, [\![\ ]\!]_{rd})$ das oben beschriebene Datenflussproblem.

zu 1. Ein Datenflussproblem ist monoton, wenn für alle Knoten n des Kontrollflussgraphen die Funktion $[\![n]\!]$ monoton ist (Definition 4.1.5).
Betrachtet man die Definition der Funktion $[\![\ ]\!]_{rd}$, so hängt diese Eigenschaft

nur von der Monotonie der Funktion $transfer_{rd} : V_{rd} \to V_{rd}$ ab. Nach Definition 5.4.4 ist die Funktion $transfer_{rd}$ monoton.
$\Rightarrow D_{rd}$ ist monoton.
$\Rightarrow$ Die MFP-Lösung von D_{rd} ist eine sichere Approximation der MOP-Lösung (Satz 4.1.1).

zu 2. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und $n \in N$ ein beliebiger Knoten.

$$\begin{aligned}
& m \in data(n, r) \\
\Rightarrow \quad & m \Downarrow_r n \\
\Rightarrow \quad & r \in def(m) \\
\wedge \quad & \exists \pi \in P[m, n] : \forall u \in \pi \backslash \{m\} : r \notin def(u) \\
\Rightarrow \quad & m \in dfi_{rd}(n)(r) \\
\Rightarrow \quad & m \in Datadep(n, r)
\end{aligned}$$

□

Somit wurde bewiesen, dass die Datenflussanalyse auf Basis der verfügbaren Definitionen eine Obermenge der tatsächlich in einem Programm vorhandenen Datenabhängigkeiten. Diese Überschätzungen kommen zum einen daher, dass die Klassifizierung der Ressourcen in der CRL-Beschreibung bereits eine Obermenge der tatsächlich betroffenen Ressourcen darstellt (vgl. Abschnitt 5.3). Zum anderen rührt diese Überschätzung aber auch von den "guarded executed"-Instruktionen. Statisch ist in den meisten Fällen nicht feststellbar, ob eine Instruktion ausgeführt wird oder nicht. Daher muss auch dafür ein konservativer Ansatz verwendet werden, der ober beschrieben wurde.

Das Datenflussproblem D_{rd} muss jetzt noch für spezielle Eigenschaften von verschiedenen Rechnerarchitekturen angepasst werden. Dies wird im folgenden Abschnitt beschrieben.

5.4.1 Behandlung von zusammengesetzten Registern

Viele Rechnerarchitekturen besitzen Register, deren einzelne Bits oder Folgen von Bits auch unter anderen Namen angesprochen werden können. Dies führt zu besonderen Problemen im Falle der Datenabhängigkeitsanalyse.

Abbildung 5.4 zeigt ein solches Register. Dabei besteht das Register R aus den beiden Register $R1$ und $R2$. Ändert man nun den Inhalt von Register R, dann ist dies auch eine Änderung an den Inhalten von Registern $R1$ und $R2$. Ändert man

dagegen den Inhalt von Register $R1$, dann ändert sich nur ein Teil von Register R. Der Inhalt von $R2$ ändert sich nicht. Dies muss in der Auswahl der Register, die von einer Instruktion betroffen sind, berücksichtigt werden.

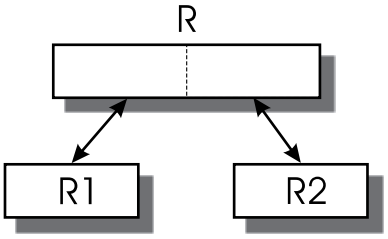


Abbildung 5.4: Zusammengesetzte Register

```
R1 = ...;
2 R2 = ...;
... = R;
```

Listing 5.2: Beispielbefehlsfolge

Betrachtet man nun die in Listing 5.2 dargestellte Befehlsfolge, so würde die Transfer-Funktion $transfer_{rd}$ folgende Datenflusswerte berechnen:

$$\begin{aligned}
pre(1): \quad & R1 \mapsto \{\} \\
& R2 \mapsto \{\} \\
& R \mapsto \{\} \\
pre(2) = post(1): \quad & R1 \mapsto \{1\} \\
& R2 \mapsto \{\} \\
& R \mapsto \{1\} \\
post(3) = pre(3) = post(2): \quad & R1 \mapsto \{1\} \\
& R2 \mapsto \{2\} \\
& R \mapsto \{2\}
\end{aligned}$$

Dies ist aber dahingehend falsch, als dass die Definition von Register $R1$ in Zeile 1 und die damit verbundene Veränderung des Registerinhaltes von Register R verloren geht.

Im allgemeinen kann man dazu folgende Lösung betrachten: Organisiert man die Register hierarchisch mittels eines Baums, dann kann man die “Teil von”-Beziehung wie folgt umsetzen. Ein Register, das Teil eines anderen, größeren Registers ist, wird in der Baumhierarchie eine Ebene tiefer angesiedelt und die beiden Register werden

miteinander verbunden. Analoges gilt für die "Besteht aus"-Beziehung. Für die Neudefinition eines Registers R an einem beliebigen Programmpunkt gilt nun folgendes:

1. Fall: Das Register R wird definitiv verändert. Dann werden alle Register, die in der Baumhierarchie Väter von R sind, eventuell verändert. Für diese dürfen also vorhandene Informationen nicht zerstört werden. Für alle Register, die in der Baumhierarchie Kinder von R sind, gilt das Gleiche wie für das Register selbst. Der Inhalt dieser Register verändert sich also ebenfalls auf jeden Fall.

2. Fall: Das Register R könnte seinen Wert verändern, d.h., der Wert des Attributs $guard = conditional$. In diesem Fall werden die Väter und Kinder dieses Registers gleich behandelt. Der Inhalt von R sowie der Väter und Kinder verändert sich eventuell, d.h., die vorhandenen Informationen dürfen an dieser Stelle nicht zerstört werden.

Dies ist in Abbildung 5.5 dargestellt.

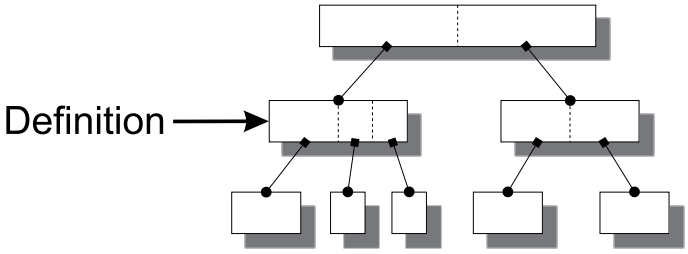


Abbildung 5.5: Registerbaum

Es ist deshalb notwendig, die Transfer-Funktion für die beiden gerade beschriebenen Probleme zu erweitern.

Definition 5.4.6 (PartOf-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die *PartOf-Funktion* $PartOf : N \rightarrow \mathcal{P}(Resource)$ für einen Knoten $n \in N$ liefert alle Ressourcen zurück, in denen eine Ressource $r \in dst(n)$ teilweise enthalten ist. Dies entspricht in der Registerbaum-Darstellung den höheren Ebenen. □

Definition 5.4.7 (Contains-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die *Contains-Funktion* $Contains : N \rightarrow \mathcal{P}(Resource)$ für einen Knoten $n \in N$ liefert alle Ressourcen zurück, die in einer Ressource $r \in dst(n)$ teilweise enthalten sind. Dies entspricht in der Registerbaum-Darstellung den tieferen Ebenen. □

Definition 5.4.8 (Transfer-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die *Transfer-Funktion* $transfer_{rd} : V_{rd} \rightarrow V_{rd}$ für einen Knoten $n \in N$ und den

betroffenen Ressourcen $R = dst(n) \cup PartOf(n) \cup Contains(n)$ ist definiert als:

$$\forall r \in R: \quad v(r) = \begin{cases} must(v, r, n), & \text{wenn } guard(n) = always \\ & \quad \wedge\, r \in dst(n) \cup Contains(n), \\ may(v, r, n), & \text{wenn } guard(n) = conditional \\ & \quad \vee\, (r \in PartOf(n) \wedge guard(n) \neq never), \\ v(r), & \text{sonst.} \end{cases}$$

$$\wedge \quad \forall r \notin R: \quad v(r) = v(r)$$

□

Mit dieser neuen Transfer-Funktion liefert das Datenflussproblem D_{rd} auch für solche Ressourcenstrukturen korrekte Ergebnisse. Die Definitionen zeigen aber auch, dass zur Realisierung eines Slicing-Algorithmus auf der Disassemblerebene die Kenntnis von Hardware-spezifischen Eigenschaften notwendig ist. So benötigt man mindestens Wissen über die Ressourcen eines Prozessors, d.h. die Register sowie die Abhängigkeit der Register untereinander. Dieses Wissen muss für jede unterstützte Hardware einzeln im Slicing-Algorithmus berücksichtigt werden. Dies kann beispielweise über einen zusätzlichen Parameter geschehen.

```
R = ...;
2 R1 = ...;
  R2 = ...;
4 ... = R;
```

Listing 5.3: Beispielbefehlsfolge

Allerdings liefert der beschriebene Ansatz teilweise Überschätzungen. Betrachtet man beispielweise die in Listing 5.3 dargestellte Beispielbefehlsfolge, so ergibt sich mit der Registerstruktur aus Abbildung 5.4 der Datenflusswert am Eingang der 4. Instruktion als:

$$pre(4) = \{1, 2, 3\}$$

Die Neudefinitionen der Register $R1$ und $R2$ in den Zeilen 2 und 3 stellen aber zusammen eine Änderung des kompletten Inhalts des Registers R dar. Der beschriebene Ansatz liefert also in diesem Fall eine Überschätzung. Korrekt wäre ein Datenflusswert von:

$$pre(4) = \{2, 3\}$$

Dies lässt sich mit diesem Ansatz allerdings nicht erreichen. Um dieses Problem zu lösen, müsste man die Register in einer Art Baum organisieren, wie er in Kapitel 6

beschrieben wird. Mit der dortigen Modellierung ist es möglich, Register bis hin zu den einzelnen Bits zu modellieren und zu updaten. Auf eine derartige Modellierung wurde hier aber verzichtet, da der Aufwand dafür zu groß gewesen wäre. In heutigen Rechnerarchitekturen können Teile eines Registers selten unter mehr als zwei Namen angesprochen werden. Darüberhinaus haben Tests gezeigt, dass eine Modellierung, wie sie hier beschrieben wurde, vollkommen ausreichend ist.

Nachdem nun gezeigt wurde, wie man die Datenabhängigkeiten eines Programms aus dem Kontrollflussgraphen ableiten kann, wird im nächsten Abschnitt beschrieben, wie man die Kontrollabhängigkeiten aus einem Kontrollflussgraphen berechnen kann.

5.5 Kontrollabhängigkeitsanalyse

Dieser Abschnitt befasst sich damit, wie man aus einem Kontrollflussgraphen durch Kombination von zwei Analysen die Kontrollabhängigkeiten eines Eingabeprogramms ableiten kann. Die erste Analyse dient dabei der Berechnung der Post-Dominator-Mengen, wie sie in Definition 5.1.8 vorgestellt wurden. Die zweite Analyse dient dann dazu, potentielle Kontrollknoten zu bestimmen. Die folgenden Abschnitte beschreiben zunächst die beiden Analysen und danach deren Kombination zur Berechnung der Kontrollabhängigkeiten.

5.5.1 Post-Dominator-Analyse

Die Post-Dominator-Analyse dient zur Umsetzung der in Definition 5.1.8 vorgestellten Post-Dominanz. Dabei sucht man für einen Knoten im Kontrollflussgraph alle Nachfolgeknoten, die auf jedem Pfad zum Endknoten enthalten sind. Als Wertebereich für die Analyse dient daher eine einfache Menge von Knoten:

$$set \equiv \mathcal{P}(N)$$

Als Operatoren auf dieser Menge dienen die Standardoperatoren $\subseteq, \cup$ und $\cap$. Da man an einem Ergebnis interessiert ist, das auf allen Pfaden gilt, handelt es sich bei der Berechnung der Post-Dominatoren um ein Schnittmengenproblem. Somit erhält man folgenden Verband als Basis für die Post-Dominator-Analyse:

$$V_{pdom} = (set, N, \emptyset, \supseteq, \cap, \cup)$$

Definition 5.5.1 (Transfer-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph Die *Transfer-Funktion* der Post-Dominator-Analyse $transfer_{pdom} : V_{pdom} \to V_{pdom}$ für einen Knoten $n \in N$ ist definiert als:

$$transfer_{pdom}(v) = v \cup \{n\}$$

□

Für die Return-Funktion $return_{pdom} : V_{pdom} \times V_{pdom} \rightarrow V_{pdom}$ gilt:

$$return_{pdom} = \cup$$

Die Abbildungsfunktion $[\![\ ]\!]_{pdom} : N \rightarrow (V_{pdom} \rightarrow V_{pdom})$ ist für einen Knoten $n \in N$ folgendermaßen definiert:

$$[\![n]\!]_{pdom} = transfer_{pdom}$$

Da man an Informationen über Knoten interessiert ist, die im Kontrollflussgraph Nachfolger des momentan betrachteten Knotens sind, ist es notwendig, die Post-Dominator-Analyse als Rückwärtsanalyse durchzuführen. Den Startknoten der Analyse initialisiert man mit der leeren Menge. Dadurch ergibt sich das Datenflussproblem D_{pdom} als:

$$D_{pdom} = (K^{-1}, V_{pdom}, [\![\ ]\!]_{pdom})$$

Beispiel 5.5.1. Betrachtet man den in Abbildung 5.6 dargestellten Kontrollflussgraph und berechnet für das Datenflussproblem D_{pdom} die MFP-Lösung, so ergibt sich:

$$\begin{aligned}
pre(3) = pre(6) = post(X) &= \{X\} \\
pre(4) = pre(5) = post(6) &= \{X, 6\} \\
post(5) &= \{X, 6, 5\} \\
post(4) &= \{X, 6, 4\} \\
post(3) &= \{X, 3\} \\
pre(2) &= \{X, 6\} \\
post(2) &= \{X, 6, 2\} \\
pre(1) &= \{X\} \\
pre(S) = post(1) &= \{X, 1\}
\end{aligned}$$

□

Die vorgestellte Analyse liefert als Ergebnis die Post-Dominator-Menge *pdom* für jeden Knoten. Wie bereits zuvor erwähnt, ist das Ergebnis notwendig, um die Kontrollabhängigkeiten eines Kontrollflussgraphen zu berechnen. Der nächste Abschnitt beschäftigt sich mit der zweiten Analyse, die benötigt wird, um zu einem Knoten alle Knoten zu bestimmen, von denen der aktuell betrachtete potentiell kontrollabhängig ist.

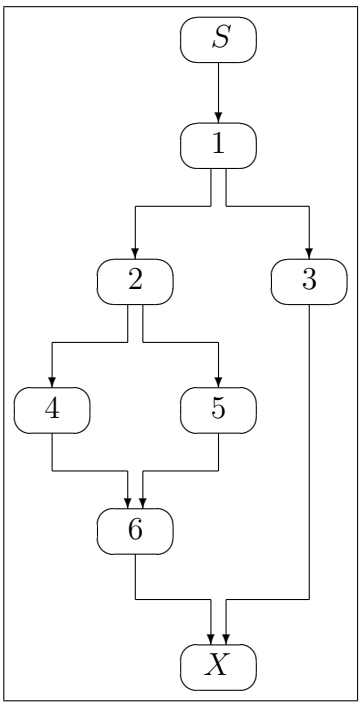


Abbildung 5.6: Beispielkontrollflussgraph

5.5.2 Kontrollpunkt-Analyse

Dieser Abschnitt stellt eine Analyse vor, die zusammen mit der in Abschnitt 5.5.1 vorgestellten Post-Dominator-Analyse dazu dient, die Kontrollabhängigkeiten aus einem Kontrollflussgraphen abzuleiten. Dazu ist es notwendig, den Begriff der Kontrolle einzuführen.

Definition 5.5.2 (Kontrolle). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Ein Knoten $i \in N$ *kontrolliert* einen Knoten $j \in N$, wenn gilt:

$$\begin{aligned} & |\{v \mid (i, v) \in E\}| \geq 2 \\ \wedge \quad & \exists \pi \in P[i, x] : j \in \pi \end{aligned}$$

□

Ein Knoten kontrolliert also einen anderen Knoten, wenn an ihm in irgendeiner Form eine Entscheidung getroffen werden kann. Im Gegensatz zur Kontrollabhängigkeit setzt diese Definition aber nicht voraus, dass es vom Kontrollknoten aus einen Pfad zum Programmende gibt, an dem der zweite Knoten nicht passiert wird. Somit stellt diese Beziehung keine Kontrollabhängigkeit zwischen diesen beiden Knoten dar.

Im Gegensatz zur Post-Dominator-Analyse handelt es sich bei der Kontrollpunkt-Analyse um eine Vorwärtsanalyse. Als Wertebereich dient hier ebenfalls die in Ab-

schnitt 5.5.1 vorgestellte Menge *set*. Um später Optimierungen realisieren zu können ergänzt man den Wertebereich um ein größtes und ein kleinstes Element $\top$ und $\bot$. Somit erhält man den Wertebereich

$$ctrldep \equiv set \cup \{\bot, \top\}$$

für die dann folgende Ordnung für gilt:

$$\begin{aligned} & \forall x \in ctrldep: \quad \bot \sqsubseteq_{ctrldep} x \sqsubseteq_{ctrldep} \top \\ \wedge \quad & x_1, x_2 \in set: \quad x_1 \subseteq x_2 \Rightarrow x_1 \sqsubseteq_{ctrldep} x_2 \end{aligned}$$

Die Operatoren $\sqcup_{ctrldep}$ und $\sqcap_{ctrldep}$ sind analog zu den Operatoren $\sqcup_{func}$ und $\sqcap_{func}$ aus Abschnitt 5.4 definiert. Für die Analyse ist man an einer Eigenschaft interessiert, die auf einem Pfad durch ein Programm gültig ist. Somit handelt es sich bei der benötigten Analyse um ein Vereinigungsproblem, das auf dem folgendem Verband V_{cp} durchgeführt wird:

$$V_{cp} = (ctrldep, \bot, \top, \sqsubseteq_{ctrldep}, \sqcup_{ctrldep}, \sqcap_{ctrldep})$$

Definition 5.5.3 (Transfer-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die *Transfer-Funktion* der Kontrollpunkt-Analyse $transfer_{cp} : V_{cp} \to V_{cp}$ für einen Knoten $n \in N$ ist definiert als:

$$transfer_{cp}(v) = \begin{cases} v \sqcup_{cp} \{n\}, & \text{wenn } |\{x \mid (n, x) \in E\}| \geq 2, \\ v, & \text{sonst.} \end{cases}$$

□

Für die Return-Funktion $return_{cp} : V_{cp} \times V_{cp} \to V_{cp}$ gilt:

$$return_{cp} = \sqcup_{ctrldep}$$

Die Abbildungsfunktion $[\![\ ]\!]_{cp} : N \to (V_{cp} \to V_{cp})$ für einen Knoten $n \in N$ ergibt sich somit als:

$$[\![n]\!]_{cp} = transfer_{cp}$$

Das Datenflussproblem D_{cp} für einen beliebigen Kontrollflussgraphen K ist dann definiert als:

$$D_{cp} = (K, V_{cp}, [\![\ ]\!]_{cp})$$

Beispiel 5.5.2. Betrachtet man den in Abbildung 5.6 dargestellten Kontrollflussgra-

phen, so ergibt sich die MFP-Lösung des Datenflussproblems D_{cp} als:

$$\begin{aligned} pre(1) = post(S) &= \emptyset \\ pre(3) = pre(2) = post(1) &= \{1\} \\ pre(5) = pre(4) = post(2) &= \{1,2\} \\ post(3) &= \{1\} \\ pre(6) = post(5) = post(4) &= \{1,2\} \\ post(6) &= \{1,2\} \\ pre(X) &= \{1,2\} \end{aligned}$$

□

Mit Hilfe dieser Analyse und der in Abschnitt 5.5.1 vorgestellten Post-Dominator-Analyse ist es nun möglich, die Kontrollabhängigkeiten eines Kontrollflussgraphen zu berechnen. Dies wird im nächsten Abschnitt beschrieben.

5.5.3 Kombination der Ergebnisse

Wie bereits zuvor erwähnt, ist ein Knoten j von einem Knoten i genau dann kontrollabhängig, wenn es mindestens einen weiteren Pfad von i zum Endknoten x gibt, auf dem j nicht enthalten ist. Zur schnellen Bestimmung der Kontrollabhängigkeiten eines Kontrollflussgraphen werden zwei Analysen verwendet. Die erste Analyse, die Post-Dominator-Analyse, dient zur Berechnung der in Definition 5.1.8 vorgestellten *pdom*-Mengen. Die zweite Analyse, die Kontrollpunkt-Analyse, dient dazu, die Anzahl der Knoten, die als Kontrollknoten in Frage kommen, einzuschränken. Kombiniert man die beiden Analyseergebnisse miteinander, so kann man für jeden Knoten des Kontrollflussgraphen die Menge der Kontrollknoten bestimmen. Somit lässt sich die Kontrollabhängigkeitsfunktion für jeden Knoten des Kontrollflussgraphen wie folgt definieren:

Definition 5.5.4 (Kontrollabhängigkeitsfunktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph, D_{pdom} und D_{cp} die oben beschriebenen Datenflussprobleme. Die Ergebnisse der Datenflussanalyse sind über die Funktionen $dfi_{pdom} : N \to V_{pdom}$ und $dfi_{cp} : N \to V_{cp}$ verfügbar. Die *Kontrollabhängigkeitsfunktion* $Ctrldep : N \to \mathcal{P}(N)$ für einen Knoten $n \in N$ ist definiert als:

$$Ctrldep(n) = \begin{cases} \emptyset, & \text{wenn } dfi_{cp}(n) = \bot \\ & \quad \vee\ dfi_{cp}(n) = \top, \\ \{m \mid m \in C : n \notin dfi_{pdom}(m)\}, & \text{wenn } dfi_{c}p(n) = C. \end{cases}$$

□

Beispiel 5.5.3. Betrachtet man wiederum den in Abbildung 5.6 dargestellten Kontrollflussgraphen und die Ergebnisse der Post-Dominator- und Kontrollpunkt-Analyse aus 5.5.1 und 5.5.2, so ergeben sich folgende Kontrollabhängigkeiten für die einzelnen Knoten:

$$\begin{aligned} Ctrldep(S) &= \emptyset \\ Ctrldep(1) &= \emptyset \\ Ctrldep(2) &= \{1\} \\ Ctrldep(3) &= \{1\} \\ Ctrldep(4) &= \{1,2\} \\ Ctrldep(5) &= \{1,2\} \\ Ctrldep(6) &= \{1\} \\ Ctrldep(X) &= \emptyset \end{aligned}$$

□

Satz 5.5.1 (Korrektheit). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Seien D_{pdom} und D_{cp} die oben beschriebenen Datenflussprobleme. Die Menge $Ctrldep(n)$ ist für alle Knoten $n \in N$ eine sichere Approximation der Kontrollabhängigkeiten und es gilt:

$$\forall n \in N : \forall m \in \mathit{infl}(n) \Rightarrow n \in Ctrldep(m)$$

□

Beweis 5.5.1. Um diesen Satz zu beweisen müssen zwei Teilaussagen bewiesen werden:

1. Die Datenflussproblem D_{pdom} und D_{cp} sind monoton $\Rightarrow MFP$-Lösung sicher.

2. Die Aussage $\forall n \in N : \forall m \in \mathit{infl}(n) : n \in CtrlDep(m)$ ist korrekt.

zu 1. Die Monotonie der beiden Datenflussprobleme folgt direkt aus der Konstruktion der beiden Transfer-Funktionen $\mathit{transfer}_{pdom}$ und $\mathit{transfer}_{cp}$. Nach Satz 4.1.1 sind daher die MFP-Lösungen der beiden Datenflussprobleme sichere Approximationen der MOP-Lösung.

zu 2. Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und sei $n \in N$ ein beliebiger Knoten.

$$
\begin{array}{lll}
 & & m \in \mathit{infl}(n) \\
\Rightarrow & & n \downarrow m \\
\Rightarrow & & \exists \pi \in P[n,m] : \forall u \in \pi \backslash \{n,m\} : m \in \mathit{pdom}(u) \\
 & \wedge & m \notin \mathit{pdom}(n) \\
\Rightarrow & & \exists \pi \in P[n,m] : \forall u \in \pi \backslash \{n,m\} : \forall \pi' \in P[u,x] : m \in \pi' \\
 & \wedge & \exists \pi'' \in P[n,x] : m \notin \pi'' \\
\Rightarrow & & |\{z \mid (n,z) \in E\}| \geq 2 \\
 & \wedge & \exists \pi \in P[n,m] \\
 & \wedge & \exists \pi'' \in P[n,x] : m \notin \pi'' \\
\Rightarrow & & n \in \mathit{dfi}_{cp}(m) \\
 & \wedge & m \notin \mathit{pdom}(n) \\
\Rightarrow & & n \in \mathit{Ctrldep}(m)
\end{array}
$$

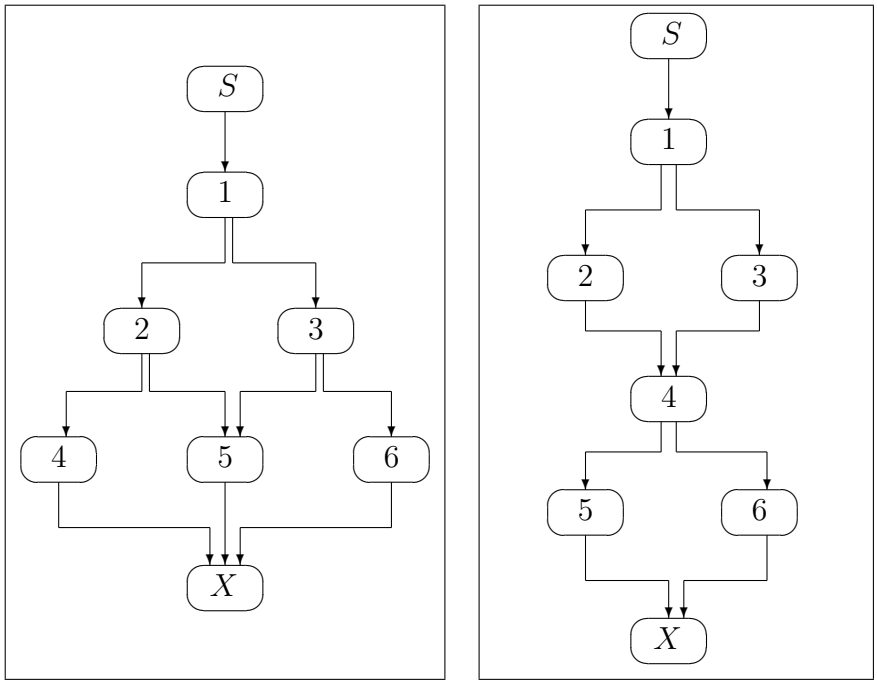
□

Nachdem nun gezeigt wurde, dass für keinen Knoten des Kontrollflussgraphen eine Kontrollabhängigkeit verloren geht, d.h. die Berechnung der Kontrollabhängigkeiten eine sichere Approximation der tatsächlich in einem Programm vorhandenen Kontrollabhängigkeiten ist, soll nun die mögliche Überschätzung des Ergebnisses näher erläutert werden.

Die in Abbildung 5.7 dargestellten Kontrollflussgraphen zeigen ein Beispiel für unstrukturierten und strukturierten Kontrollfluss. Es ist leicht zu sehen, dass der Graph in Abbildung 5.7(a) nicht ohne Sprung-Befehle aus einer Hochsprache erzeugt werden kann. Da aber in dieser Arbeit nicht davon ausgegangen werden kann, dass die zu analysierenden Programme aus Hochsprachen entstanden sind, müssen die Analysen auch für solche Beispiele korrekte Ergebnisse liefern.

Die gerade vorgestellte Kombination der Analysen liefert für dieses Beispiel korrekte Ergebnisse. Würde man beispielsweise anstelle der Kontrollpunkt-Analyse eine Dominator-Analyse verwenden, dann würde bei zusammenfließendem Kontrollfluss der Schnitt über die eingehenden Informationen gebildet werden. Damit würde man dann aber auch die beiden Kontrollpunkte 2 und 3 am Programmpunkt 5 verlieren. Somit wäre für diesen Fall das Analyseergebnis falsch. Damit würden dann auch falsche Slices berechnet.

Betrachtet man nun den in Abbildung 5.7(b) dargestellten strukturierten Kontrollflussgraphen, so würde an dieser Stelle eine Dominator-Analyse, kombiniert mit den Ergebnissen der Post-Dominator-Analyse, korrekte Abhängigkeiten liefern. Dagegen liefert die hier vorgestellte Kontrollpunkt-Analyse an dieser Stelle eine Überschätzung



(a) unstrukturierter Kontrollfluss (b) strukturierter Kontrollfluss

Abbildung 5.7: Beispiele für strukturierten und unstrukturierten Kontrollfluss

der Abhängigkeiten: Kontrollknoten 1 ist auch in der Kontrollpunktmenge der Knoten 5 und 6 enthalten, obwohl zwischen diesen Knoten keine Kontrollabhängigkeit besteht. Die Kombination der Ergebnisse würde beispielweise für den Knoten 5 die Knoten 1 und 4 als Kontrollknoten liefern. Dies ist eine Überschätzung, die sich nur mit erheblichem Mehraufwand beseitigen lässt.

Experimente haben aber gezeigt, dass es in nur sehr wenigen Fällen zu einer derartigen Überschätzung kommt (vgl. Kapitel 7.2). Somit ist der Mehraufwand, der betrieben werden müsste, um solche Fälle auszuschließen, an dieser Stelle nicht gerechtfertigt und wird auch in dieser Arbeit nicht weiter beschrieben. Wichtiger für den Slicing-Algorithmus sind Ergebnis, die in jedem Fall korrekt, bzw. sichere Approximationen darstellen. Dies ist, wie bereits bewiesen wurde, in jedem Fall gegeben.

Der nun folgende Abschnitt zeigt, wie durch Kombination der drei gerade beschriebenen Analysen ein Slice für ein beliebiges Slicing-Kriterium berechnet werden kann.

```
Eingabe: Kontrollflussgraph K = (N, E, s, x)

löse D_rd für KFG K
löse D_pdom für KFG K
löse D_cp für KFG K

solange kein Abbruch gewünscht {
  warte auf neues Slicing−Kriterium C = (n, V)
  workset = {(n, v) | v ∈ C}
  visited = ∅
  solange (workset ≠ ∅) {
    sei (m, w) ∈ workset
    visited = visited ∪ {(m, w)} ∪ {(c, _) | c ∈ Ctrldep(m)}
    workset = workset\{(m, w)} ∪ ((⋃_{u∈src(m)} {(x, u) | x ∈ Datadep(m, u)}
              ∪ ⋃_{o∈Ctrldep(m),u∈src(o)} {(x, u) | x ∈ Datadep(o, u)}) ∩ visited)
  }
  slice = {m | (m, w) ∈ visited}
}
```

Listing 5.4: Slicing-Algorithmus

5.6 Slicing-Algorithmus

Nachdem in den beiden vorhergehenden Abschnitten, 5.4 und 5.5, die Daten- und Kontrollabhängigkeitsanalysen vorgestellt wurden und deren Korrektheit bewiesen wurde, wird in diesem Abschnitt ein iterativer Algorithmus zum statischen Rückwärts-Slicing vorgestellt. Dieser orientiert sich dabei eng an dem in Abschnitt 5.1 vorgestellten iterativen Algorithmus zur Berechnung der indirekt relevanten Anweisungen S_C^{k+1}.

Zunächst werden für einen Kontrollflussgraph **einmal** die beschriebenen Analysen durchgeführt. Die Ergebnisse dieser Analysen werden anschließend für den Algorithmus benötigt und in temporären Datenstrukturen gespeichert. Danach kann für beliebige Slicing-Kriterien ein Slice berechnet werden. Der dazu erforderliche Algorithmus ist in Listing 5.4 schematisch dargestellt.

Als Eingabe erhält der Algorithmus einen beliebigen Kontrollflussgraphen in Form einer CRL-Datei. Dann werden zunächst die Datenflussprobleme gelöst und somit die Abhängigkeiten der Ressourcen und der Programmpunkte, bzw. der Knoten untereinander bestimmt.

Danach können beliebig oft für gegebene Slicing-Kriterien neue Slices berechnet werden. Für ein gegebenes Kriterium wird wie folgt der Slice berechnet:
Der Algorithmus verwaltet zwei Mengen von Tupeln aus Knoten und Ressource. Die-

se können auch als Slicing-Kriterien aufgefasst werden, bei denen nur eine Ressource betrachtet wird. Eine Menge (*workset*), bestehend aus Tupeln (Knoten × Ressource), beinhaltet alle Programmpunkte, die noch näher betrachtet werden müssen. Die Menge *visited* speichert alle Elemente der Menge *workset*, die schon einmal betrachtet wurden. Sie dient zum einen dazu, jeden Programmpunkt für eine spezielle Ressource nur einmal zu betrachten und garantiert somit die Terminierung des Algorithmus. Zum anderen stellt die Projektion der Elemente dieser Menge auf die Knoten des Kontrollflussgraphen den Slice hinsichtlich des gegebenen Kriteriums dar.

Die Menge der noch zu betrachtenden Programmpunkte *workset* wird mit allen Ressourcen des Slicing-Kriteriums initialisiert. Danach wird solange iteriert, bis die Menge *workset* leer ist.

In jeder Iteration wird ein Element aus der Menge entfernt. Das entnommene Element wird mit seinen Kontrollabhängigkeiten zur Menge der bereits betrachteten Elemente *visited* hinzugefügt. Die direkten Datenabhängigkeiten des entnommenen Elementes sowie die der Kontrollknoten werden wiederum zur Menge der noch zu betrachtenden Programmpunkte hinzugefügt, allerdings werden dabei die bereits betrachteten Elemente vorher entfernt.

Als Ergebnis dieser Schleife erhält man den Slice hinsichtlich des gegebenen Slicing-Kriteriums, indem man die Elemente der Menge *visited* auf die Knoten des Kontrollflussgraphen abbildet. Wie man leicht sehen kann, ist dieser Algorithmus eine leicht modifizierte Umsetzung der indirekt relevanten Anweisungen S_C^{k+1}. Wie bereits erwähnt wurde, stellt der Fixpunkt der Folge $\left(S_C^{k+1}\right)_{k\in\mathbb{N}}$ einen sicheren Slice bezüglich eines gegebenen Slicing-Kriteriums ist. Der modifizierte Algorithmus terminiert spätestens nach $|N| \times |Resource|$ vielen Iterationen, da zu diesem Zeitpunkt alle möglichen Elemente, die in dem Kontrollflussgraphen vorkommen können, in der Menge *visited* der bereits betrachteten Elemente enthalten sind. Somit werden dann auch keine neuen Elemente mehr in die Menge *workset* eingefügt.

Da für einen Kontrollflussgraphen die Kontroll- und Datenabhängigkeiten unabhängig von einem spezifischen Slicing-Kriterium vorberechnet werden, kann der oben beschriebene Vorgang beliebig oft für einen Kontrollflussgraphen wiederholt werden.

Aufbauend auf diesem Algorithmus und den drei Analysen können die Ergebnisse noch verbessert und verfeinert werden. Eine Beschreibung von verschiedenen Optimierungen folgt im nächsten Abschnitt.

5.7 Optimierungen

Dieser Abschnitt beschreibt einige Optimierungen, die dazu dienen, die Datenflussanalysen zum einen exakter, zum anderen aber auch schneller zu machen. Dazu kann man

z.B. alle nicht-erreichbaren Pfade von den Analysen ausschließen oder den Algorithmus um semantische Informationen der einzelnen Instruktionen erweitern. Gerade die Optimierungen, die auf semantischem Wissen beruhen, machen die Analysen sehr viel präziser. Dadurch lassen sich viele Ungenauigkeiten, die, wie in Abschnitt 5.4 beschrieben, in der Datenabhängigkeitsanalyse auftreten, erheblich verbessern. Die folgenden Abschnitte beschreiben einen Auswahl von Optimierungen, die in dieser Form auch implementiert wurden.

5.7.1 Nicht-erreichbare Pfade

Ein nicht-erreichbarer Pfad ist ein Teilpfad innerhalb eines Kontrollflussgraphen, von dem statisch bestimmt werden kann, dass er in keinem Fall zur Ausführungzeit des Programms ausgeführt wird. Somit sind alle Anweisungen auf einem solchen Teilpfad für die Ergebnisse, die das Programm berechnet, uninteressant und müssen daher auch nicht in einem Slice berücksichtigt werden. Nicht-erreichbare Pfade lassen sich mit Hilfe einer besonderen Analyse, der Werte-Analyse, bestimmen ([Sic97] und [FKL+99]).

Definition 5.7.1 (*infeasible*-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die Funktion $\mathit{infeasible} : N \to \mathbb{B}$ mit $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ ist die Funktion, die jedem Knoten $n \in N$ zuordnet, ob das dazugehörende Programmfragment $F(n)$ bei der Ausführung des Programms gemäß den Ergebnissen der Werte-Analyse möglicherweise (*false*) oder sicher nicht ausgeführt wird (*true*). □

Benutzt man die Ergebnisse dieser Analyse bei der Berechnung der Daten- und Kontrollabhängigkeiten eines Programms, so lassen sich hierdurch bessere Ergebnisse erzielen. Dazu ist es nur notwendig, die Transfer-Funktionen der einzelnen Analysen dahingehend anzupassen, dass auf solchen Pfaden keine Datenflusswerte mehr berechnet werden. Im Falle der hier durchgeführten Analysen müssen dazu die Transfer-Funktionen der Datenflussprobleme D_{rd}, D_{pdom} und D_{cp} angepasst werden. Zunächst werden nun die Änderungen an der Transfer-Funktion der Datenabhängigkeitsanalyse D_{rd} beschrieben.

Dabei handelt es sich um eine Vorwärtsanalyse auf dem Wertebereich *func*, der ein größtes und ein kleinstes Element $\top, \bot$ besitzt. Da bei dieser Analyse eine Eigenschaft interessant ist, die auf mindestens einem Pfad gilt, wird bei zusammenfließendem Kontrollfluss die Vereinigung $\sqcup_{\mathit{func}}$ der eingehenden Datenflussinformationen gebildet. Um keine Informationen zu verlieren, wäre es daher falsch, die Datenflusswerte auf einem nicht-erreichbaren Pfad auf $\top$ zu setzen. Somit würde man die Datenflusswerte an einem Vereinigungsknoten zerstören. Setzt man auf allen nicht-erreichbaren Pfaden die Datenflusswerte auf $\bot$, bleiben die berechneten Informationen an Vereinigungsknoten dagegen erhalten. Daraus ergibt sich die Transfer-Funktion $\mathit{transfer}_{rd}$ der Datenabhängigkeitsanalyse D_{rd} für einen Knoten $n \in N$ eines Kontrollflussgraphen $K =$

(N, E, s, x) und den betroffenen Ressourcen $R = dst(n) \cup PartOf(n) \cup Contains(n)$ als:

$$\forall r \in R: \quad v(r) = \begin{cases} \bot, & \text{wenn } infeasible(n) = true, \\ must(v, r, n), & \text{wenn } guard(n) = always \\ & \quad \wedge\, r \in dst(n) \cup Contains(n), \\ may(v, r, n), & \text{wenn } guard(n) = conditional \\ & \quad \vee\, (r \in PartOf(n) \wedge guard(n) \neq never), \\ v(r), & \text{sonst.} \end{cases}$$

$$\wedge \quad r \notin R: \quad v(r) = v(r)$$

Eine ähnliche Überlegung kann man jetzt auch für die Berechnung der Kontrollabhängigkeiten anstellen. Ein Knoten n kann nach Definition 5.5.4 nur dann von einem Knoten m kontrollabhängig sein, wenn n nicht in der Post-Dominator-Menge von m enthalten ist. Setzt man nun die Datenflusswerte auf allen nicht-erreichbaren Pfaden auf die Menge N aller Knoten des Kontrollflussgraphen, so ist genau diese Eigenschaft nicht mehr erfüllt. Der Knoten n ist dann nicht mehr von einem nicht-erreichbaren Knoten kontrollabhängig. Die erweiterte Transfer-Funktion $transfer_{pdom}$ für die Post-Dominator-Analyse sieht daher für einen Kontrollflussgraphen $K = (N, E, s, x)$ wie folgt aus:

$$transfer_{pdom}(v) = \begin{cases} N, & \text{wenn } infeasible(n) = true, \\ v \cup \{n\}, & \text{sonst.} \end{cases}$$

Da bei diesem Datenflussproblem bei mehreren eingehenden Kanten der Schnitt über die Datenflusswerte gebildet wird, ist auch hier garantiert, dass keine Informationen verloren gehen.

Für die Kontrollpunktanalyse D_{cp} ist es ebenfalls notwendig, die Transfer-Funktion anzupassen. Da bei diesem Problem bei mehreren eingehenden Kanten in einen Knoten die Vereinigung der eingehenden Informationen gebildet wird, ist es daher notwendig, den Datenflusswert auf allen nicht-erreichbaren Pfaden auf $\bot$ zu setzen. Dadurch ergibt sich die Transfer-Funktion $transfer_{cp}$ für einen Kontrollflussgraphen $K = (N, E, s, x)$ als:

$$transfer_{cp}(v) = \begin{cases} \bot, & \text{wenn } infeasible(n) = true, \\ v \sqcup_{cp} \{n\}, & \text{wenn } |\{x \mid (n, x) \in E\}| \geq 2, \\ v, & \text{sonst.} \end{cases}$$

Das Ergebnis dieser Änderungen sind exaktere Vorhersagen über die Daten- und Kontrollabhängigkeiten eines Programms, die man mit Hilfe einer vorher durch-

geführten Werte-Analyse erhalten hat. Durch diese Optimierung kann in vielen Fällen der Fixpunkt in den Datenflussanalysen wesentlich schneller berechnet werden. Durch das Ausschließen von Pfaden werden aber auch die Ergebnisse des Slicings erheblich verbessert. Ein Pfad, der ohnehin in keiner Ausführung des Programms durchlaufen wird, beeinflusst durch diese Optimierung auch nicht mehr die Ergebnisse der Datenflussanalysen. Tauchen diese Instruktionen nicht mehr in diesen Ergebnissen auf, dann sind sie auch in keinem Fall in einem Slice für ein beliebiges Kriterium enthalten. Somit dient diese Optimierung zum einen zur Beschleunigung, zum anderen aber auch der qualitativen Verbesserung der Slices.

5.7.2 Semantische Informationen

Dieser Abschnitt beschreibt zwei mögliche Optimierungen, die man mit Hilfe von prozessorspezifischem Wissen durchführen kann. Allerdings bleibt dabei anzumerken, dass Optimierungen unter Verwendung von semantischen Informationen hardwarespezifisch sind und der resultierende Algorithmus daher für unterschiedliche Prozessoren angepasst werden muss.

5.7.2.1 Aufruf-Konventionen

Für viele Prozessoren gelten feste Regeln, an die sich ausführbare Programme halten müssen. So gibt es unter anderem Einschränkungen, welche Register die Parameter für einen Funktionsaufruf enthalten, sowie in welchem Register der Rückgabewert gespeichert werden muss. Diese Art der Einschränkung wird als **Aufruf-Konvention** bezeichnet. Dies kann für Optimierungen verwendet werden: Die Register einer Architektur werden in zwei Teilmengen aufgespalten. Diese Mengen sind disjunkt voneinander und werden wie folgt gebildet: Eine Menge beinhaltet alle Register, die vor einem Funktionsaufruf vom Aufrufer gesichert werden müssen, die andere Menge beinhaltet alle Register, die von der aufgerufenen Funktion gesichert werden müssen.

Die Kenntnis dieser Mengen kann bei der Berechnung der verfügbaren Definitionen benutzt werden. Alle Register, die von der aufgerufenen Prozedur gesichert werden müssen, werden beim Verlassen dieser wieder restauriert. Somit ist sichergestellt, dass die Werte der entsprechenden Register nach einem Prozeduraufruf genau gleich den Werten der Register vor dem Prozeduraufruf sind. Dies ist schematisch in Abbildung 5.8 dargestellt. Hierbei ist r_1 ein Register, dass über Prozeduraufrufe hinweg seinen Wert behält, wohingegen r_2 nach einem Prozeduraufruf einen geänderten Wert enthalten kann.

Diese Aufruf-Konventionen lassen sich, wenn sie bekannt sind, in der Datenabhängigkeitsanalyse mit betrachten. Dazu ist es notwendig, die Return-Funktion

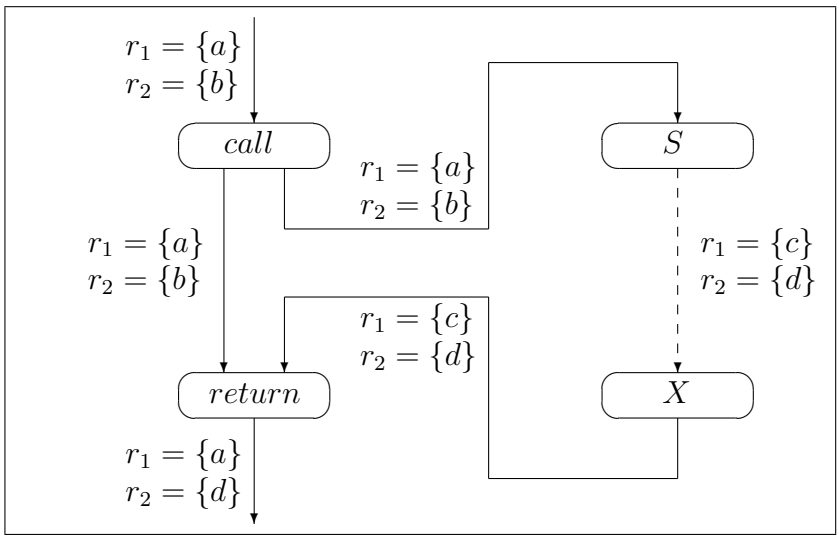


Abbildung 5.8: Aufruf-Konventionen

für $return_i$-Knoten zu verändern. Die Menge der Register, die vom Aufrufer gespeichert werden, wird im folgenden mit $CalleeSaved$ bezeichnet. Betrachtet man einen Knoten $return_i \in N$ eines Kontrollflussgraphen, so gilt für die Return-Funktion $return_{rd} : V_{rd} \times V_{rd} \to V_{rd}$ für eine spezielle Prozessorarchitektur:

$$return(v_{call_i}, v_{x_i})(r) = \begin{cases} v_{call_i}(r), & \text{wenn } r \in CalleeSaved, \\ v_{x_i}(r), & \text{wenn } r \in Resource \backslash CalleeSaved. \end{cases}$$

An diesem Punkt ist bereits bekannt, dass eine erneute Definition eines dieser Register $r \in Resource \backslash CalleeSaved$ die ursprüngliche Definition verdeckt. Dagegen ist für alle Register $r \in CalleeSaved$ bekannt, dass der ursprüngliche Wert des Registers vor der Rückkehr der aufgerufenen Funktion wieder hergestellt wird. Mit dieser Erweiterung der Datenabhängigkeitsanalyse lässt sich nun genau der in Abbildung 5.8 schematisch dargestellte Sachverhalt realisieren. Durch diese Anpassung lassen sich nun exaktere Slices berechnen, allerdings benötigt man dafür prozessorspezifisches Wissen. Dieses Wissen kann dem Algorithmus über einen Parameter übergeben werden und ist auch nur innerhalb der Datenabhängigkeitsanalyse interessant.

Allerdings gelten diese Aufruf-Konventionen nur für Funktionen, die mittels eines Compilers aus einer Hochsprache erzeugt wurden. Da aber die meisten Bibliotheksfunktionen nicht aus der Hochsprache direkt erzeugt wurden, sondern meistens direkt in Assemblercode geschrieben sind, müssen die geforderten Konventionen dort nicht gelten. Oftmals wird innerhalb dieser Bibliotheksfunktionen bewusst dieses spezifische Wissen "missbraucht", um so noch effizienteren und kleineren Code zu realisieren. Es ist daher notwendig, die beschriebene Optimierung in solchen Funktionen nicht

anzuwenden. Da sich ein Compiler neben diesen Aufruf-Konventionen auch an Namenskonventionen für eine generierte Funktion hält und diese sich eindeutig von den Namenskonventionen innerhalb einer Bibliothek unterscheiden, können die Funktionen dadurch unterschieden werden. In den meisten Fällen gilt dabei, dass eine Bibliotheksfunktion mit zwei Unterstrichen "__" beginnt, während für Funktionen, die aus der Hochsprache übersetzt werden, null bis ein Unterstrich verwendet wird.

Wodurch die Funktionen sich unterscheiden lässt sich ebenfalls über einen Parameter in die Analysen einbringen. Somit kann die gerade beschriebene Optimierung ohne Probleme in das bestehende Framework integriert werden. Optimierungen dieser Art dienen der Verbesserung der Slicing-Ergebnisse.

5.7.2.2 Semantik von Instruktionen

Eine weitere Optimierung, die an der Datenabhängigkeitsanalyse ansetzt, ist das Hinzufügen von semantischen Informationen für einzelnen Instruktionen. Betrachtet man z.B. die folgende Instruktion des ARM-Prozessors

```
stmia r13! {r0,r1,r2};
```

die einen Zugriff auf den Stack des Programms ausführt, dort die drei Register $r0$, $r1$ und $r2$ speichert und anschließend das Stackpointer Register $r13$ um 12 byte verändert.

Die CRL-Beschreibung dieser Instruktion sieht wie folgt aus:

```
"stmia r13!, {r0,r1,r2}": src1="r13", dst1="r13", src2="r0", src3="r1",
                          src4="r2", dst5="Mem", guard="always";
```

Berechnet man einen Slice mit dem Algorithmus aus Abschnitt 5.6, dann würde sich z.B. für das Register $r13$ folgende Abhängigkeit ergeben: Der neue Wert von $r13$ hängt von den Registerinhalten von $r13$, $r0$, $r1$ und $r2$ ab. Dies ist aber eine unnötige Überschätzung der tatsächlichen Abhängigkeiten, da die Werte der Register $r0$, $r1$ und $r2$ nur auf dem Stack gespeichert werden, nicht aber den neuen Wert des Stackpointer Registers $r13$ beeinflussen.

Allgemein ist es zur Verbesserung der Slicing-Ergebnisse sinnvoll, die Semantik der einzelnen Instruktionen zu kennen. Dazu ist es notwendig, die $src()$-Funktion aus Abschnitt 5.3 um einen zusätzlichen Parameter, nämlich die betrachtete Ressource, zu erweitern. Daraus ergibt sich dann die folgenden Funktion für einen Knoten $n \in N$ eines Kontrollflussgraphen und eine Ressource $r \in Resource$:

$$src(n, r) = \{src_i(n) \mid i \in I\!N \wedge src_i(n) \text{ beeinflusst } r\}$$

Durch die exaktere Kenntnis der Abhängigkeiten der CRL-Attribute voneinander für die verschiedenen Instruktionen lässt sich mit dieser Methode das Ergebnis des Slicings noch weiter verbessern. Die daraus resultierenden Slices enthalten unter Umständen deutlich weniger Knoten und verbessern somit die Qualität erheblich. Die dafür nötigen Anpassungen betreffen aber nicht das Slicing-Framework, sondern können über eine geeignete Schnittstelle in den Algorithmus eingebracht werden. Dazu kann man sich beispielweise eine geeignete Sprache zur Beschreibung der Quell-Ziel-Abhängigkeit vorstellen. Dadurch ist zur Anpassung an eine neue Architektur lediglich eine solche Beschreibung neu zu erstellen. Eine Optimierung dieser Art dient ebenfalls der qualitativen Verbesserung der Slicing-Ergebnisse.

Kapitel 6

Modellierung von Speicherzugriffen

Nachdem in Kapitel 5 der grundlegende Slicing-Algorithmus und die benötigten Analysen vorgestellt wurden, beschäftigt sich dieses Kapitel mit einer Ressource, die in der bisherigen Diskussion noch nicht erwähnt wurde: dem Speicher.

Zunächst wird das Problem von Speicherzugriffen und ein allgemeines Speichermodell beschrieben. Anschließend folgt eine Beschreibung eines dynamischen Modells, das zur Modellierung von Speicherzugriffen verwendet und in dieser Form auch implementiert wurde. Danach wird eine Datenflussanalyse zur Umsetzung des Speichermodells und die Benutzung dieser Informationen zur Berechnung von Slices beschrieben. Abschließend wird auf mögliche Optimierungen eingegangen.

6.1 Problembeschreibung

Die bisherige Diskussion des Slicing-Algorithmus und der benötigten Analysen hat eine zentrale Ressource moderner Rechnerarchitekturen nicht betrachtet. Bei dieser Ressource handelt es sich um den Speicher, in dem z.B. Daten abgelegt werden, die aus Platzgründen nicht mehr in Registern des Prozessors gehalten werden können. Welche Werte zu welchem Zeitpunkt in den Speicher ausgelagert werden, und welche Werte in Registern gehalten werden, wird dabei vom Compiler entschieden. Neben solchen Werten sind der *Stack* und *Heap* eines Programms im Speicher angesiedelt.

Betrachtet man den in Listing 6.1(a) dargestellten Beispielcode, so stellt sich für die Berechnung eines Slices folgendes Problem: Würde man den Speicher als eine einzige, große Ressource ansehen, so werden die einzelnen Zellen des Speichers nicht getrennt voneinander betrachtet. Berechnet man für das Kriterium $C = (5, R_1)$ einen Slice, so ergibt sich mit dem momentanen Algorithmus das in Listing 6.1(b) dargestellte falsche Teilprogramm. Ein konservativer Slice ist dagegen in Listing 6.1(c) dargestellt. Für dessen Berechnung betrachtet man alle Schreibzugriffe in den Speicher als mögliche Veränderung der kompletten Ressource - ähnlich zu der in Abschnitt 5.4.1 vorgestellten Behandlung von zusammengesetzten Registern. In großen Eingabeprogrammen mit vielen Speicherzugriffen führt dies aber zu unbefriedigenden Ergebnissen. Interessiert

	(a)	(b)	(c)	(d)
1	$M[0] = 1;$		$M[0] = 1;$	$M[0] = 1;$
2	$M[1] = 2;$	$M[1] = 2;$	$M[1] = 2;$	
3	$R_1 = M[0];$	$R_1 = M[0];$	$R_1 = M[0];$	$R_1 = M[0];$
4	$R_2 = M[1];$			
5	$R_1 = R_1 + R_2;$	$R_1 = R_1 + R_2;$	$R_1 = R_1 + R_2;$	$R_1 = R_1 + R_2;$
6	$M[0] = R_1;$			

Listing 6.1: Beispielprogramm, falscher, konservativer und minimaler Slice

ist man, wie zuvor bereits erwähnt, an möglichst kleinen Slices, d.h. an Slices, die möglichst wenige Anweisungen enthalten. Ein solcher minimaler Slice ist in Listing 6.1(d) für das obige Kriterium dargestellt.

Wie dieses Beispiel zeigt, ist es für die Qualität eines Slices, d.h. die Anzahl der Anweisungen, wichtig, den Speicher und damit auch Speicherzugriffe in einer Art zu modellieren, die es gestattet, verschiedene Zugriffe zu unterscheiden. Dazu benötigt man eine Speicherfunktion Ξ, die wie folgt definiert ist:

Definition 6.1.1 (Speicherfunktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph und *Addr* die Menge der möglichen Speicheradressen einer Rechnerarchitektur. Eine *Speicherfunktion* $\Xi : Addr \to \mathcal{P}(N)$ ist eine Abbildung, die jeder Adresse $a \in Addr$ die Menge der Knoten zuordnet, an den der dazugehörende Speicherinhalt zum letzten mal verändert wurde. □

Eine Veränderung eines Speicherinhalts führt dann dazu, dass die Speicherfunktion Ξ ebenfalls verändert werden muss. Dazu definiert man die Funktionen *must* und *may*. Diese unterscheiden sich dadurch, dass *must* für eine gegebene Adresse alle bisherigen definierenden Vorkommen löscht, während *may* die Menge der letzten Definitionen nur vergrößert.

Definition 6.1.2 (*must*-Update). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die Funktion $must : (Addr \to \mathcal{P}(N)) \times N \times \mathcal{P}(Addr) \to (Addr \to \mathcal{P}(N))$ ist wie folgt definiert:

$$must(f, n, I)(a) = \begin{cases} n, & \text{wenn } a \in I, \\ f(a), & \text{wenn } a \notin I. \end{cases}$$

Dabei ist $I \subseteq \mathcal{P}(Addr)$ ein Intervall, das die Grenzen des Speicherzugriffs beschreibt. □

Definition 6.1.3 (*may*-Update). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die Funktion $may : (Addr \to \mathcal{P}(N)) \times N \times \mathcal{P}(Addr) \to (Addr \to \mathcal{P}(N))$ ist wie folgt

definiert:

$$may(f, n, I)(a) = \begin{cases} f(a) \cup n, & \text{wenn } a \in I, \\ f(a), & \text{wenn } a \notin I. \end{cases}$$

Dabei ist $I \subseteq \mathcal{P}(Addr)$ ein Intervall, das die Grenzen des Speicherzugriffs beschreibt. □

Mit Hilfe dieser Abbildung Ξ und den beiden Update-Funktionen *must* und *may* ist es nun möglich, jede Speicherzelle, die durch ihre Adresse repräsentiert ist, einer Menge von Programmpunkten zuzuordnen, an denen der Inhalt dieser Speicherzelle verändert wurde.

Dieses Konzept kann in einem trivialen Ansatz genau so implementiert werden. Eine solche Implementierung ist allerdings sehr ineffizient, was vor allem an der Größe des zu modellierenden Speichers liegt. Den kompletten Speicher zu modellieren ist unnötig, da Programme meist nur auf einen kleinen, zumeist überschaubaren Teil des insgesamt verfügbaren Speichers zugreifen. Andererseits wird bei einen Zugriff meistens nicht nur eine Speicherzelle verändert, sondern mehrere aufeinanderfolgende Zellen auf einmal. Daher ist es für eine Implementierung sinnvoll, solche Zugriffe gemeinsam zu behandeln.

Leider ist es nicht möglich, den Speicher vorab in sinnvolle Teile so zu zerlegen, dass man die unterschiedlichen Speicherzugriffe getrennt voneinander modellieren kann. Somit ist es sinnvoll, ein Modell des Speichers zu entwickeln mit dem Speicherzugriffe und deren Zugriffsbreite dynamisch modelliert werden können. Ein solches Modell wird im folgenden Abschnitt vorgestellt.

6.2 Modellbeschreibung

Dieser Abschnitt beschreibt ein dynamisches Speichermodell auf Basis einer Baumstruktur, mit der sich die verschiedenen Zugriffe in den Speicher voneinander unterscheiden lassen. Als Ergebnis lassen sich dann Slices berechnen, die weniger Anweisungen enthalten als es ein konservativer Ansatz liefern würde und somit qualitativ besser sind.

Als Basis für das Speichermodell dient eine einfache binäre Baumstruktur, d.h. jeder Knoten des Baums ist entweder ein *Blatt* und hat keine Nachfolger, oder der Knoten hat genau zwei Nachfolger. Diese werden dann als *innere Knoten* bezeichnet. Zusätzlich ist jedem Knoten ein Intervall zugeordnet, das die Speichergrenzen wiedergibt. Ein Blatt besitzt außerdem eine Menge, in der Knoten eines Kontrollflussgraphen gesammelt werden - genau diejenigen, an denen der Speicherbereich, der durch das Intervall beschrieben wird, verändert wurde.

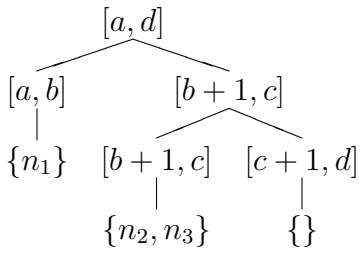


Abbildung 6.1: Beispielspeicherbaum

Definition 6.2.1 (Speicherbaum). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die Menge M der *Speicherbäume* wird rekursiv wie folgt definiert: Ein Tupel $(x_1, x_2, A) \in L$ mit $L \equiv \mathbb{N} \times \mathbb{N} \times \mathcal{P}(N)$ heißt *Blatt*. Ein Tupel $(x_1, x_2, t_1, t_2) \in I$ mit $I \equiv \mathbb{N} \times \mathbb{N} \times M \times M$ heißt *innerer Knoten*. Ein Element $m \in L|I$ heißt *Speicherbaum*, wenn gilt:

$$
\begin{array}{lll}
\text{Fall 1:} & & \text{Sei } m = (x_1, x_2, t_1, t_2) \in I \\
& & \text{o.B.d.A. } t_1 = (y_1, y_2, ...) \wedge t_2 = (w_1, w_2, ...) \\
& & x_1 < x_2 \\
& \wedge & y_1 = x_1 \\
& \wedge & w_2 = x_2 \\
& \wedge & y_1 \leq y_2 < w_1 \leq w_2 \\
& \wedge & y_2 = w_1 - 1 \\
& \wedge & t_1, t_2 \in M \\
\text{Fall 2:} & & \text{Sei } m = (x_1, x_2, A) \in L \\
& & x_1 \leq x_2 \\
& \wedge & A \subseteq N
\end{array}
$$

□

Betrachtet man beispielsweise den Speicherbaum, der in Abbildung 6.2 dargestellt ist und für den die Ordnung $a < b < c < d$ gilt, so bedeutet dies, dass die Speicherzellen in dem Intervall $[a, b]$ an der Programmstelle n_1, die Speicherzellen im Bereich $[b+1, c]$ an den Programmstellen n_2 und n_3 verändert wurden. Der Bereich $[c + 1, d]$ wurde noch nicht verändert. Dies entspricht einer Abbildung der Speicherfunktion Ξ, die wie folgt aussieht, wenn im Kontrollflussgraphen n_1 vor n_2 und n_2 vor n_3 betrachtet wird:

$$may(\Xi, n_3, \{b + 1, \ldots, c\}) \circ must(\Xi, n_2, \{b + 1, \ldots, c\} \circ must(\Xi, n_1, \{a, \ldots, b\})$$

Beginnt man nun initial mit dem Element $m = (0, \infty, \emptyset) \in L$, das den leeren Speicher darstellt, so kann man darauf verschiedene Transformationsfunktionen definieren, die Elemente aus M in neue Elemente aus M transformieren. Dazu muss

zunächst unterschieden werden, ob die Grenzen eines Speicherzugriffs statisch exakt zu bestimmen sind oder nicht. Im ersten Fall werden die betroffenen Speicherzellen definitiv verändert, wohingegen im letzteren Fall nur ein Teil der Speicherzellen verändert wird. Dieser Effekt ist ähnlich zu dem in Abschnitt 5.4.1 vorgestellten Problem von zusammengesetzten Registern. Im Falle des Speicherbaums ist es dabei notwendig, vorhandene Elemente an den Blättern nicht zu zerstören.

Im folgenden werden nun verschiedene Transformationen beschrieben, mit denen ein Speicherbaum, der aus dem initialen Baum m entstanden ist, verändert werden kann. Dazu definiert man die beiden Funktionen $must : M \times N \times \mathbb{N} \times \mathbb{N} \to M$ und $may : M \times \mathcal{P}(N) \times \mathbb{N} \times \mathbb{N} \to M$, die einen Speicherbaum an einem Knoten $n \in N$ eines Kontrollflussgraphen $K = (N, E, s, x)$ transformieren. Dabei geht man davon aus, dass es eine Funktion $addr : N \to (\mathbb{N} \times \mathbb{N})$ gibt, die für einen gegebenen Knoten $n \in N$ eines Kontrollflussgraphen die linke und rechte Grenze des Speicherzugriffs zurückliefert. Zur Definition der beiden Funktionen *must* und *may* muss zwischen Updates von Blättern und inneren Knoten eines Speicherbaumes unterschieden werden.

6.2.1 Update eines Blattes

Zunächst sollen nun die möglichen Transformationen an Blättern des Speicherbaumes näher beschrieben werden. Dazu betrachtet man o.B.d.A das folgende Blatt $(a, d, P) \in M$:

$$\begin{array}{c}[a,d]\\ |\\ P\end{array}$$

Für einen beliebigen Knoten n eines Kontrollflussgraphen $K = (N, E, s, x)$, dessen zugeordnetes Programmfragment $F(n)$ den Speicher verändert, müssen folgende Fälle unterschieden werden: Dabei wird davon ausgegangen, dass die Grenzen des Speicherzugriffs immer innerhalb der Grenzen des Blattes liegen.

1. Fall: Die Adresse des nächsten Zugriffs stimmt mit den Grenzen des Baumes überein, d.h. $addr(n) = (a, d)$.

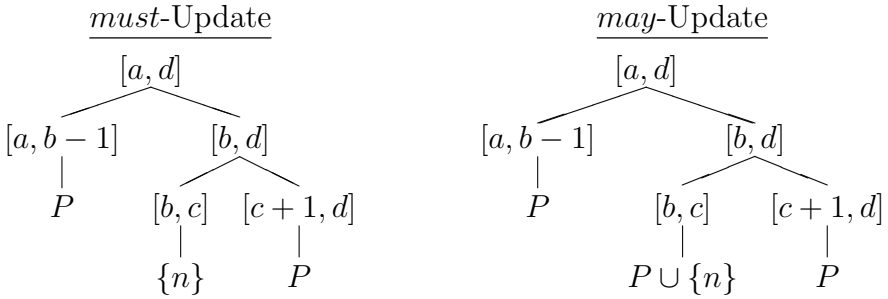
$$\underline{must\text{-Update}} \qquad\qquad \underline{may\text{-Update}}$$

$$\begin{array}{c}[a,d]\\ |\\ \{n\}\end{array} \qquad\qquad \begin{array}{c}[a,d]\\ |\\ P \cup \{n\}\end{array}$$

2. Fall: Die Adresse des Zugriffs stimmt mit der linken Grenze des gegebenen Blattes überein, d.h. $addr(n) = (a, b)$ mit $b < d$.

3. Fall: Die Adresse des Zugriffs stimmt mit der rechten Grenze des Baums überein, d.h. $addr(n) = (c, d)$ mit $a < c$.

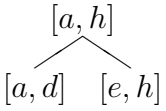
4. Fall: Die Adresse des Speicherzugriffs liegt innerhalb der Grenzen des Blattes, d.h. $addr(n) = (b, c)$ mit $a < b \leq c < d$.



Mit Hilfe dieser Definition lässt sich für einen Speicherzugriff aus einem Speicherbaum ein neuer Speicherbaum bilden, der den Speicherzugriff korrekt modelliert. Der nachfolgende Abschnitt beschreibt die Definition der Funktionen $must()$ und $may()$ für innere Knoten eines Speicherbaums.

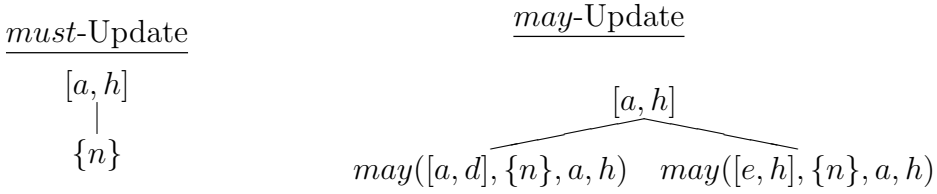
6.2.2 Update eines inneren Knoten

Dabei kann man allgemein von folgender Ausgangssituation für einen inneren Knoten eines Speicherbaums (a, h, t_1, t_2) ausgehen. Dabei gilt o.B.d.A. $t_1 = (a, d, ...)$ und $t_2 = (e, h, ...)$:

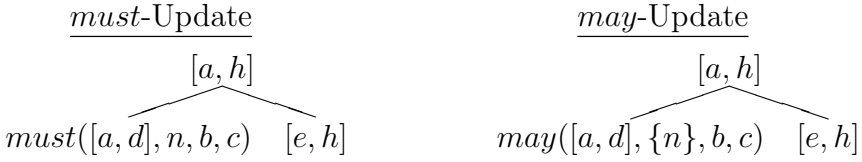


Zur Definition der *must*-Funktion für innere Knoten benötigt man zusätzlich die beiden Funktionen $plb : M \times \mathbb{N} \to M$ und $prb : M \times \mathbb{N} \to M$, die am Ende dieses Abschnittes definiert sind. Die Update-Funktionen $must()$ und $may()$ für einen Knoten $n \in N$ eines Kontrollflussgraphen $K = (N, E, s, x)$, dessen zugeordnetes Programmfragment $F(n)$ den Speicher verändert, sind wie folgt definiert:

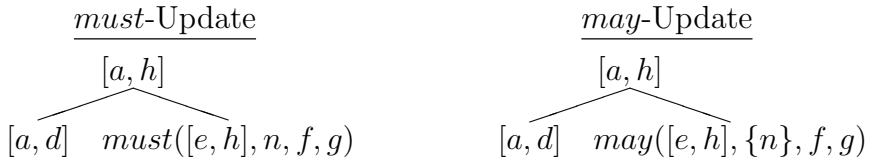
1. Fall: Die Grenzen des Speicherzugriffs stimmen mit den Grenzen des inneren Knoten überein, d.h. $addr(n) = (a, h)$.



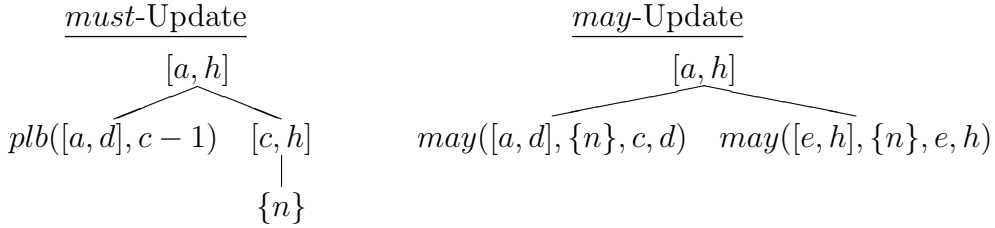
2. Fall: Die Grenzen des Speicherzugriffs sind innerhalb der Grenzen des linken Kindes t_1 des Speicherbaums, d.h. $addr(n) = (b, c)$ mit $a \leq b \leq c \leq d$.



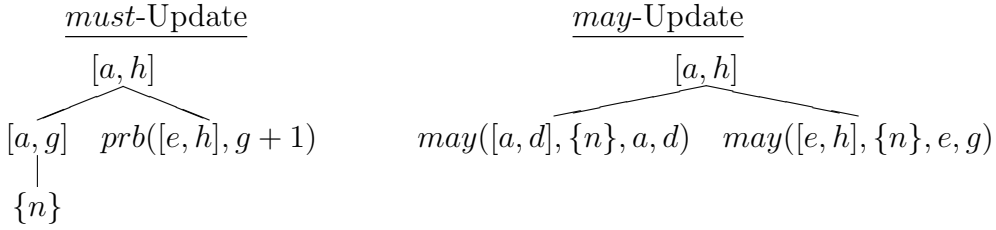
3. Fall: Die Grenzen des Speicherzugriffs sind innerhalb der Grenzen des rechten Kindes t_2 des Speicherbaums, d.h. $addr(n) = (f, g)$ mit $e \leq f \leq g \leq h$.



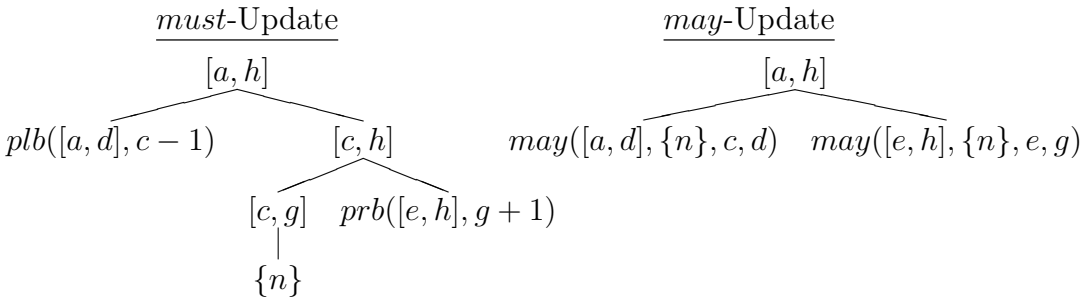
4. Fall: Der Speicherzugriff geht über die Grenzen des rechten Kindes t_2 hinaus und stimmt mit der rechten Grenze des inneren Knoten überein, d.h. $addr(n) = (c, h)$ mit $a < c <= d$.



5. Fall: Der Speicherzugriff geht über die Grenzen des linken Kindes t_1 hinaus und stimmt mit der linken Grenze des inneren Knoten überein, d.h. $addr(n) = (a, g)$ mit $e \leq g < h$.

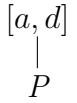


6. Fall: Der Speicherzugriff liegt innerhalb der Grenzen des inneren Knotens und geht über die Grenzen der Kinder t_1 und t_2 hinaus, d.h. $addr(n) = (c, g)$ mit $a < c \leq d < e \leq g < h$.



Mit Hilfe dieser Definitionen von *must* und *may* lässt sich der Speicher, bzw. lassen sich die Speicherzugriffe dynamisch modellieren. Im folgenden werden noch die beiden Funktionen $plb : M \times \mathbb{N} \to M$ und $prb : M \times \mathbb{N} \to M$ definiert. Die Funktion *plb* dient dabei dazu, eine gegebene Zahl als neue rechte Grenze des gegebenen Speicherbaums einzusetzen, für die Funktion *prb* gilt analoges für die linke Grenze. Zur Definition muss hierbei ebenfalls zwischen Blättern und inneren Knoten unterschieden werden. Darüberhinaus ist zu beachten, dass die gegebene Grenze innerhalb der Grenzen des Speicherbaums liegen muss.

1. Fall: Der gegebene Speicherbaum m ist ein Blatt und sieht allgemein wie folgt aus:



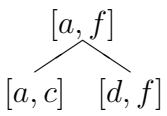
Für diesen Fall sieht die Definition der beiden Funktionen für eine neue Grenze b wie folgt aus:

```
    R = 0;
2   beg :  jump R > 5 end;
    M[R] = 1;
4   R = R + 1;
    jump beg;
6   end :
```

Listing 6.2: Beispielbefehlsfolge für zyklisches Schreiben in den Speicher

$$\underline{plb(m,b)} \qquad \underline{prb(m,b)}$$
$$[a,b] \qquad [b,d]$$
$$| \qquad |$$
$$P \qquad P$$

2. Fall: Ist der gegebene Speicherbaum m ein innerer Knoten, d.h. $m \in I$, dann kann von folgendem allgemeinen Baum ausgegangen werden:



Dabei gilt $t_1 = (a, c, ...)$ und $t_2 = (d, f, ...)$. In diesem Fall muss zusätzlich unterschieden werden, ob die neue Grenze innerhalb der Grenzen von t_1 oder innerhalb der Grenzen von t_2 liegt. Liegt die neue Grenze b innerhalb der Grenzen von t_1, so sehen die Transformationsfunktionen wie folgt aus:

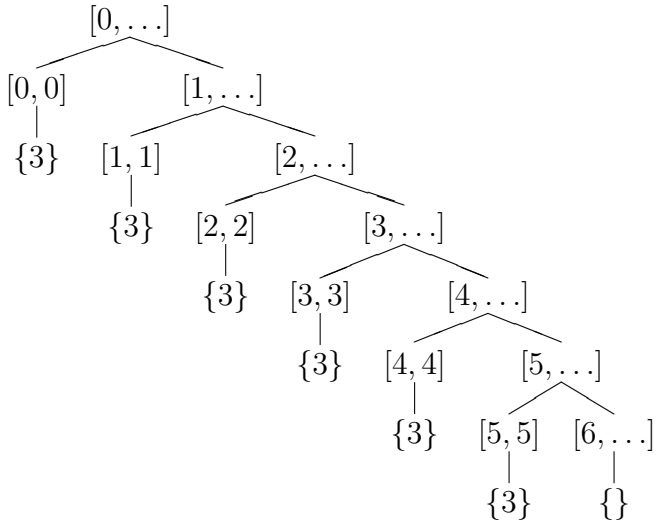
$$\underline{plb(m,b)} \qquad \underline{prb(m,b)}$$
$$plb([a,c],b) \qquad [b,f]$$
$$prb([a,c],b) \quad [d,f]$$

Liegt die Grenze e innerhalb der Grenzen von t_2, so sehen die Transformationsfunktionen wie folgt aus:

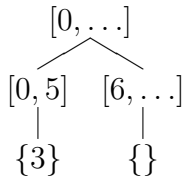
$$\underline{plb(m,e)} \qquad \underline{prb(m,e)}$$
$$[a,e] \qquad prb([d,f],e)$$
$$[a,c] \quad plb([d,f],e)$$

Mit Hilfe dieser Funktionen lassen sich nun Speicherbäume in neue Speicherbäume transformieren. Diese Funktionen können aber je nach Eingabeprogramm anstelle ausgeglichener Bäume lange Ketten erzeugen. Daher kann man sich die Frage stellen, ob

es sinnvoll ist, diese Speicherbäume zu balancieren. Listing 6.2.2 zeigt ein Beispielprogramm, bei dem in einer Schleife in aufeinanderfolgende Adressbereiche geschrieben wird. Wenn man die einzelnen Zugriffe in den Speicher getrennt voneinander behandelt, ergibt sich dadurch folgender Speicherbaum:



In der statischen Analyse lässt sich ein solches Verhalten aber nur dann erreichen, wenn man die Anzahl der Iterationen ermitteln kann. Dann kann diese Schleife entsprechend ihrer Iterationen ausgerollt werden und man würde diesen Effekt feststellen. Da in vielen Fällen die Schleifengrenze nicht feststellbar ist, werden diese Zugriffe zumeist gemeinsam behandelt. In diesem Beispiel bedeutet dies, dass der Adressbereich $[0, 5]$ an der Stelle 3 verändert wird. Somit würde ein Speicherbaum erzeugt werden, der wie folgt aussieht:



Da die meisten Programme aber nur auf einen kleinen Teil des insgesamt verfügbaren Speichers zugreifen, bleibt es aber fraglich, ob sich durch eine Balancierung effizientere Bäume generieren lassen oder ob der dafür notwendige Mehraufwand diese Effizienzsteigerung zunichte macht. In Einzelfällen, wie dem obigen Beispiel, kann es zu einer Verbesserung führen, in den meisten Fällen ist aber nicht damit zu rechnen. Aus diesem Grund wurde hier auf eine tiefergehende Diskussion einer möglichen Balancierung verzichet.

Der folgende Abschnitt zeigt nun, wie mit Hilfe eines Speicherbaums und der gerade beschriebenen Funktionen eine Datenflussanalyse modelliert werden kann, deren Datenflusswerte den statisch ermittelbaren Inhalt des Speichers wiedergeben.

6.3 Speicheranalyse

Dieses Kapitel beschreibt, wie mit Hilfe des, im vorherigen Abschnitt 6.2 vorgestellten, Speicherbaums und einer externen Werte-Analyse ein Datenflussproblem formuliert werden kann, dessen Datenflusswerte den statisch ermittelbaren Speicherinhalt zur Ausführung des betrachteten Programmpunktes beinhalten.

Basis für diese Analyse sind die Ergebnisse einer zuvor durchgeführten Werte-Analyse ([Sic97], [TMSS03] und [FKL+99]). Diese Analyse dient zum einen dazu, nicht-erreichbare Pfade im Kontrollflussgraphen eines Programms zu identifizieren, zum anderen aber auch, um den Inhalt der verschiedenen Prozessor-Ressourcen über die abstrakte Ausführung des Programms hinweg zu modellieren. Die Ergebnisse einer solchen Analyse lassen sich dann über verschiedene Funktionen abfragen und dienen als Basis der im folgenden beschriebenen Datenflussanalyse.

Definition 6.3.1 (*addr*-Funktion, *ambiguous*-Funktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph. Die Funktion $addr : N \to (\mathbb{N} \times \mathbb{N})$ ordnet jedem Knoten $n \in N$ einen Adressbereich des Speichers zu, in den das Programmfragment $F(n)$ bei der Ausführung des Programms eventuell zugreift. Die Funktion $ambiguous : N \to \mathbb{B}$ mit $\mathbb{B} = \{true, false\}$ ordnet jedem Knoten $n \in N$ zu, ob der Adressbereich $addr(n)$ eindeutig bestimmt werden konnte oder nicht, d.h. ob das Programmfragment $F(n)$ definitiv in den Adressbereich $addr(n)$ zugreift oder nur in einen Teil davon. □

Die Funktion $addr()$ liefert dabei einem Knoten $n \in N$ den Addressbereich zurück, in den das dazugehörende Programmfragment $F(n)$ eventuell zugreift. Dabei liefert die Funktion immer den ganzen Adressbereich zurück, d.h. es kann in einem Speicherbaum niemals die Situation auftreten, dass zwei benachbarte Blätter die gleichen Blattmengen besitzten. Somit ist es nicht notwendig, bei der Modellierung eine Funktion zum Zusammenführen von Blättern in Speicherbäumen vorzusehen.

Als Wertebereich für die folgende Speicheranalyse dient der in Definition 6.2.1 vorgestellte Speicherbaum, also die Menge M. Diese versieht man mit einem zusätzlichen größten und kleinsten Element $\top$ und $\bot$, so dass sich der Wertebereich für die Analyse wie folgt ergibt:

$$mem \equiv M \cup \{\bot, \top\}$$

Für diesen Wertebereich gilt des weiteren folgende Ordnung $\sqsubseteq_{mem}$:

$$\forall m \in mem : \quad \bot \sqsubseteq_{mem} m \sqsubseteq_{mem} \top$$

Für die Vereinigungsfunktion $\sqcup_{mem}$ gilt dann:

$$l \sqcup_{mem} m = \begin{cases} may(l, P, a, b), & \text{wenn } m = (a, b, P) \in L, \\ (l \sqcup_{mem} t_1) \sqcup_{mem} t_2, & \text{mit } m = (a, b, t_1, t_2) \text{ sonst.} \end{cases}$$

Die Schnittmengenfunktion $\sqcap_{mem}$ wird in diesem Datenflussproblem nicht benötigt und kann daher ignoriert werden. Somit ergibt sich der Verband V_{mem} als:

$$V_{mem} = (mem, \bot, \top, \sqsubseteq_{mem}, \sqcup_{mem}, \sqcap_{mem})$$

Definition 6.3.2 (Transfer-Funktion). Sei $K = (N, E, s, x)$ der Kontrollflussgraph einer CRL-Datei. Die *Transfer-Funktion* der Speicheranalyse $transfer_{mem} : V_{mem} \rightarrow V_{mem}$ für einen Knoten $n \in N$ mit $addr(n) = (a, b)$ ist definiert als:

$$transfer_{mem}(m) = \begin{cases} (0, \infty, \emptyset), & \text{wenn } n = s, \\ m, & \text{wenn } Mem \notin dst(n), \\ must(m, n, a, b), & \text{wenn } infeasible(n) = false \\ & \quad \wedge\, ambiguous(n) = false, \\ may(m, n, a, b), & \text{wenn } infeasible(n) = false \\ & \quad \wedge\, ambiguous(n) = true, \\ \bot, & \text{sonst.} \end{cases}$$

□

Für die Return-Funktion $return_{mem} : V_{mem} \times V_{mem} \rightarrow V_{mem}$ gilt:

$$return_{mem} = \sqcup_{mem}$$

Dann kann man die Abbildungsfunktion $[\![\ ]\!]_{mem} : N \rightarrow (V_{mem} \rightarrow V_{mem})$ für einen Knoten $n \in N$ wie folgt definieren:

$$[\![n]\!]_{mem} = transfer_{mem}$$

Daraus lässt sich dann das folgende Datenflussproblem D_{mem} für einen Kontrollflussgraphen $K = (N, E, s, x)$ formulieren:

$$D_{mem} = (K, V_{mem}, [\![\ ]\!]_{mem})$$

Beispiel 6.3.1. Betrachtet man beispielweise den in Abbildung 6.2 dargestellten Kontrollflussgraphen, dann würde sich für die letzte Instruktion der in Abbildung 6.3 dargestellte Datenflusswert der Speicheranalyse ergeben, wenn der Stackpointer auf die Adresse $0x80$ zeigt. Die dargestellten Instruktionen sind Instruktionen der ARM-Architektur ([ARM00] und [ARM99]). □

Nun kann man eine Funktion $GetDefs : M \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{P}(N)$ definieren, die für einen gegebenen Speicherbaum $m \in M$ und einer linken und rechten Grenze (a, b)

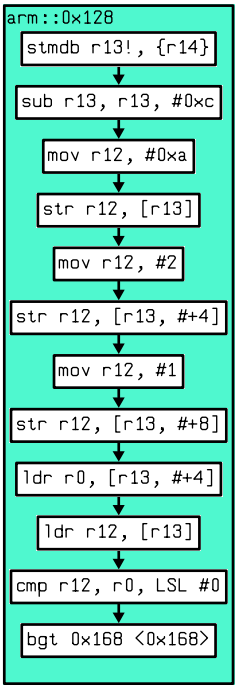


Abbildung 6.2: Beispielkontrollflussgraph

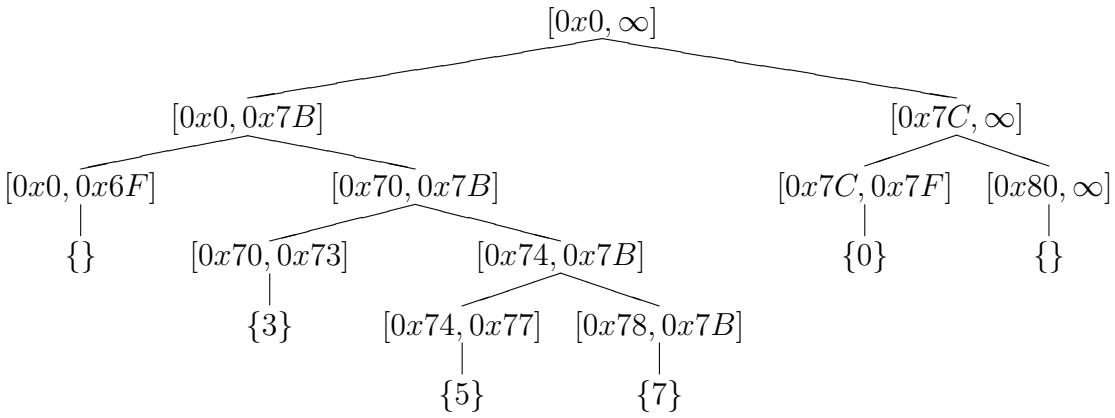


Abbildung 6.3: Datenflusswert der Speicheranalyse aus Abbildung 6.2

die Vereinigung der Knotenmengen aller Blätter berechnet, deren Grenzen ganz oder teilweise in der gegebenen Grenze enthalten sind. Für ein Blatt $(x_1, x_2, P) \in L$ definiert man diese Funktion also wie folgt:

$$GetDefs((x_1, x_2, P), a, b) = \begin{cases} P, & \text{wenn } x_1 \leq a \leq x_2 \\ & \quad \vee\, x_1 \leq b \leq x_2 \\ & \quad \vee\, a \leq x_1 \leq x_2 \leq b, \\ \emptyset, & \text{sonst.} \end{cases}$$

Für einen inneren Knoten (x_1, x_2, t_1, t_2) definiert man die Funktion wie folgt:

$$GetDefs((x_1, x_2, t_1, t_2), a, b) = GetDefs(t_1, a, b) \cup GetDefs(t_2, a, b)$$

Damit kann man die Ergebnisse der Speicheranalyse dieses Datenflussproblems abfragen, indem man die Funktion $Memdep()$ wie folgt definiert:

Definition 6.3.3 (Speicherzuordnungsfunktion). Sei $K = (N, E, s, x)$ ein Kontrollflussgraph, D_{mem} das oben beschriebene Datenflussproblem und $dfi_{mem} : N \to V_{mem}$ die Funktion, die zu einem Knoten die Lösung des Datenflussproblems D_{mem} zurückliefert. Die *Speicherzuordnungsfunktion* $Memdep : N \to \mathcal{P}(N)$ für einen Knoten $n \in N$ ist definiert als:

$$Memdep(n) = \begin{cases} GetDefs(dfi_{mem}(n), a, b), & \text{wenn } Mem \in src(n) \\ & \quad \wedge\, addr(n) = (a, b), \\ \emptyset, & \text{sonst.} \end{cases}$$

□

Mit Hilfe dieser Funktion lässt sich nun der Slicing-Algorithmus aus Kapitel 5.6 auf Speicherzugriffe erweitern. Dies wird im folgenden Abschnitt beschrieben.

6.4 Erweiterter Slicing-Algorithmus

Dieser Abschnitt beschreibt den um die Speicheranalyse erweiterten Slicing-Algorithmus, mit dessen Hilfe sich korrekte Slices auch bezüglich von Speicherzugriffen berechnen lassen. Dabei wird der in Listing 5.4 vorgestellte Algorithmus leicht modifiziert. Diese modifizierte Fassung ist in Listing 6.3 schematisch dargestellt.

```
Eingabe: Kontrollflussgraph K = (N, E, s, x),
         Ergebnis der Werte-Analyse
```

```
löse D_rd für KFG K
löse D_pdom für KFG K
löse D_cp für KFG K
löse D_mem für KFG K

solange kein Abbruch gewünscht {
   warte auf neues Slicing Kriterium C = (n, V)
   workset = {(n, v) | v ∈ V}
   visited = ∅
   solange (workset ≠ ∅) {
      sei (m, w) ∈ workset
      visited = visited ∪ {(m, w)} ∪ {(c, _) | c ∈ Ctrldep(m)}
      workset = workset \ {(m, w)} ∪ ((⋃_{u∈src(m)\{Mem}} ({(x, u) | x ∈ Datadep(m, u)}
                ∪ {(x, Mem) | x ∈ Memdep(m)})
                ∪ ⋃_{o∈Ctrldep(m), u∈src(o)\{Mem}} ({(x, u) | x ∈ Datadep(o, u)}
                      ∪ {(x, Mem) | x ∈ Memdep(o)})) ∩ visited)
   }
   slice = {m | (m, w) ∈ visited}
}
```

Listing 6.3: Erweiterter Slicing-Algorithmus

Die einzige Änderung bezüglich der in Abschnitt 5.6 vorgestellten Version besteht in der Berechnung der Tupel, die noch betrachtet werden müssen. Dabei wird nun, wenn die Ressource Speicher betrachtet werden soll, nicht mehr das Ergebnis der Datenabhängigkeitsanalyse verwendet, sondern das Ergebnis der Speicheranalyse. Allerdings hängt die Qualität der Speicheranalyse dabei sehr stark von den Ergebnissen der vorher durchgeführten Werte-Analyse ab. Je exakter deren Ergebnis, desto exakter lässt sich der Speicher mittels des Speicherbaums modellieren.

Bevor ein Slice berechnet werden kann, müssen nun vier verschiedene Analysen durchgeführt werden. Danach kann für den gleichen Kontrollflussgraph beliebig oft ein neuer Slice berechnet werden.

Nachdem ein Slicing-Kriterium ausgewählt wurde, wird rückwärts ausgehend vom gewählten Programmpunkt der Slice berechnet. Dazu wird zunächst der ausgewählte Programmpunkt mit den gewählten Ressourcen in die Menge *workset* aufgenommen. Die Menge der bereits betrachteten Tupel *visited* wird mit der leeren Menge initialisiert.

Danach wird für jedes Element in der Menge *workset* iteriert. Das aktuell entnommene Element wird zur *visited*-Menge hinzugefügt, alle Daten- und Kontrollabhängigkeiten sowie die Speicherabhängigkeiten werden in die Liste der noch zu betrachteten Tupel aufgenommen.

Die Schleife wird verlassen, wenn sich kein weiteres Element mehr in der Menge *workset* befindet. Der gesuchte Slice entspricht dann der Projektion der Elemente der *visisted*-Menge auf die Knoten des Kontrollflussgraphen. Auch dieser erweiterte Algorithmus terminiert nach $|N| \times |Resource|$ vielen Iterationen, allerdings muss nun eine weitere Analyse im Vorfeld durchgeführt werden.

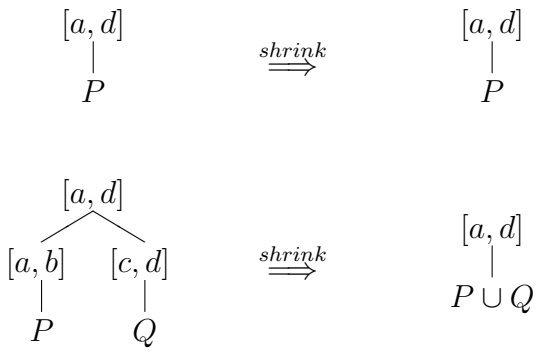
Die Ergebnisse der Speicheranalyse, die hier zur Qualitätssteigerung des berechneten Slices verwendet werden, können je nach Eingabekontrollflussgraph recht unterschiedlich sein. Der folgende Abschnitt beschreibt einige Optimierungen, mit denen sich die Berechnung der Datenflusswerte der Speicheranalyse beschleunigen lässt.

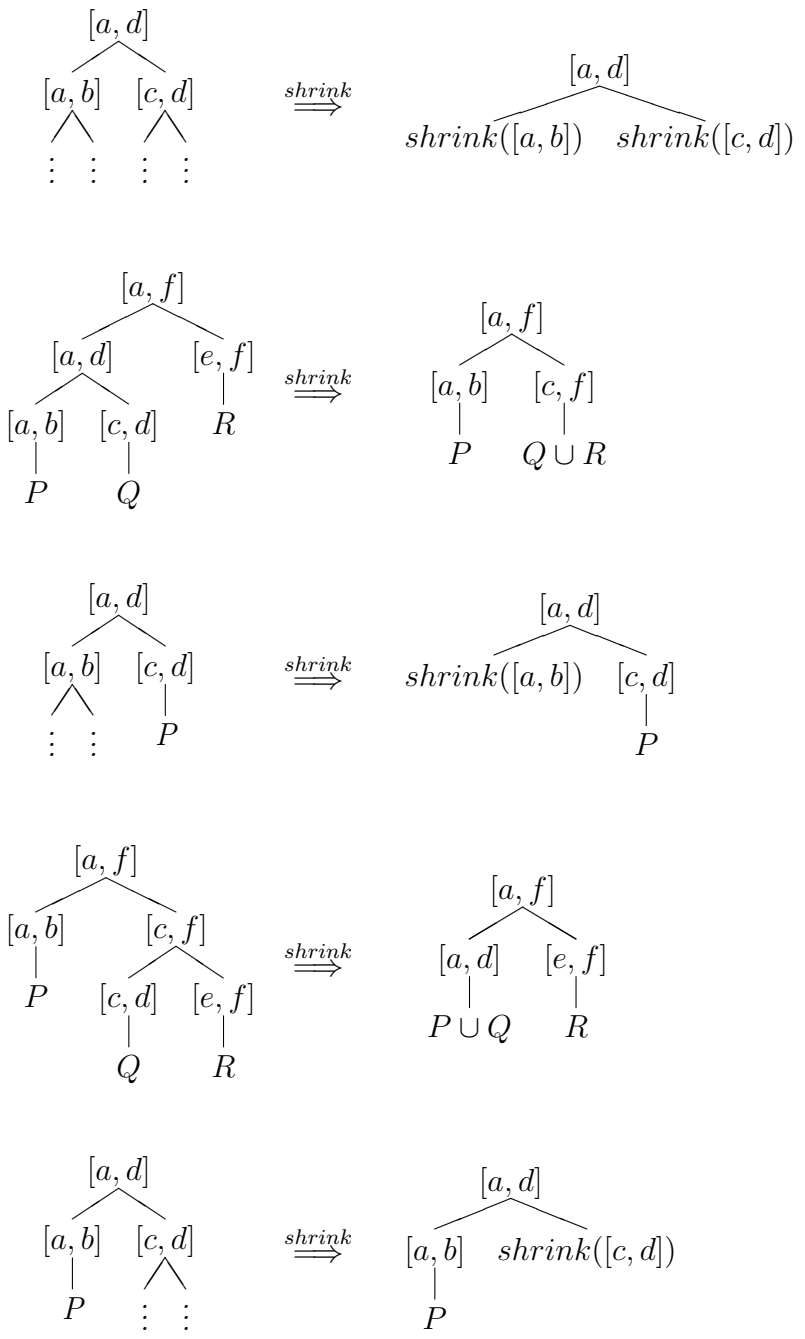
6.5 Optimierungen

Je nach Größe des Eingabekontrollflussgraphen und der Genauigkeit der Ergebnisse der Werte-Analyse, kann die Durchführung der Speicheranalyse viel Zeit in Anspruch nehmen oder sehr große Speicherbäume produzieren. Um eine praktische Umsetzung der beschriebenen Analyse zu gewährleisten, ist es daher sinnvoll, trotz möglicher Informationsverluste ein Widening für die Speicheranalyse zu definieren. Bei ungenauen Ergebnissen der Werte-Analyse können dagegen verschiedene Speicherzellen von vielen Programmpunkten beeinflusst werden, obwohl dieses Verhalten bei der wahren Ausführung des Eingabeprogramms sich nicht bestätigt. Auch in diesem Fall macht es Sinn, trotz Informationsverluste die Speicheranalyse zu beschleunigen. Dieser Abschnitt beschreibt diese beiden Optimierungen.

6.5.1 Widening auf Speicherbäumen

Zur Beschleunigung von Datenflussanalysen kann das in Kapitel 4.1.2.4 vorgestellte Widening verwendet werden. Im Falle der Speicheranalyse definiert man sich dazu zunächst die folgende Funktion $shrink : M \to M$:





Für alle anderen Fälle liefert die Funktion *shrink*() einfach die Identität, also $shrink(m) = m$, zurück. Damit lässt sich ein Widening-Operator für die Speicheranalyse D_{mem} wie folgt definieren:

Definition 6.5.1 (Widening-Funktion). Sei $D_{mem} = (K, V_{mem}, [\![\ ]\!]_{mem})$ das oben beschriebene Datenflussproblem. Die *Widening-Funktion* $\bigtriangledown_{mem} : (V_{mem} \times V_{mem} \rightarrow$

V_{mem}) ist definiert als:

$$\bigtriangledown_{mem}(m_1, m_2) = \begin{cases} \top, & \text{wenn } m_1 = \top \vee m_2 = \top, \\ m_2, & \text{wenn } m_1 = \bot, \\ m_1, & \text{wenn } m_2 = \bot, \\ shrink(m_1 \sqcup_{mem} m_2), & \text{sonst.} \end{cases}$$

□

Verwendet man diesen Widening-Operator bei der Lösung des Datenflussproblems D_{mem}, so erhält man auf Kosten eines Informationsverlustes - man bildet die Vereinigung der Informationen von zwei benachbarten Blättern - den Fixpunkt schneller. Die Ergebnisse der Datenflussanalyse sind aber nach wie vor korrekt, bzw. sicher, da kein Element verloren geht. Es werden lediglich für eine Speicherzelle unter Umständen zu viele Programmpunkte zurückgeliefert.

Eine weitere Form der Optimierung der Speicheranalyse ist im folgenden Abschnitt dargestellt.

6.5.2 Mengenoptimierung

Dieser Abschnitt beschreibt eine Optimierung der Speicheranalyse durch partiellen Informationsverlust an den Blättern. Teilweise sind die Ergebnisse der vorgeschalteten Werte-Analyse recht unbefriedigend, nämlich immer dann, wenn beispielsweise ein Register mit einem Wert initialisiert ist, den man statisch nicht ermitteln kann. Wird dieser Registerinhalt als Basis oder Offset für einen Speicherzugriff verwendet, dann lässt sich das genaue Ziel dieses Zugriffs nicht bestimmen. Um korrekt zu bleiben, muss man also davon ausgehen, dass die Instruktion irgendwo in den Speicher schreibt. Die Werte-Analyse liefert in diesem Fall den ganzen Adressbereich des Speichers als Grenzen des Zugriffs zurück, das *ambiguous*-Flag wird dabei aber auf *true* gesetzt. Nach Definition der Transfer-Funktion für die Speicheranalyse würde der Knoten des Kontrollflussgraphen, zu dem die Instruktion gehört, in die Mengen eines jeden Blattes eingefügt werden. Treten solche Zugriffe in einem Programm gehäuft auf, so führt dies zu einem schnellen Wachstum der Blattmengen eines Speicherbaums. Gerade für die praktische Umsetzung ist es daher notwendig, die Größe des Baumes, aber auch die Anzahl der Elemente an einem Blatt überschaubar zu halten.

Dazu verändert man die Funktion $may : M \times N \times \mathbb{N} \times \mathbb{N} \rightarrow M$ dahingehend, dass bei jeder Vereinigung $P \cup \{n\}$ an einem Knoten $n \in N$ eines Kontrollflussgraphen zunächst überprüft wird, ob die Anzahl der Elemente in der Menge eine vorher definierte Anzahl überschreitet. Falls dies der Fall ist, entfernt man alle Elemente aus der Menge und ersetzt sie durch ein Pseudo-Element $\mp$.

Um dann korrekte Slices berechnen zu können, ist es notwendig, eine Auflistung aller Speicherzugriffe des Programms bis zum jeweiligen Ausführungszeitpunkt zu berechnen. Dazu verändert man einfach die Transfer-Funktion der Datenflussabhängigkeitsanalyse. Anstatt für die Ressource Speicher wie bisher zwischen *may*- und *must*-Update zu unterscheiden, was, wie bereits gezeigt, sowieso zu einem falschen Slice führt, werden nun einfach alle Programmpunkte, die den Speicher verändern, aufgesammelt. Die erweiterte Transfer-Funktion $transfer_{rd}$ der Datenabhängigkeitsanalyse D_{rd} für einen Knoten $n \in N$ eines Kontrollflussgraphen $K = (N, E, s, x)$ und den betroffenen Ressourcen $R = dst(n) \cup PartOf(n) \cup Contains(n)$ ist dann definiert als:

$$\forall r \in R : v(r) = \begin{cases} \bot, & \text{wenn } infeasible(n) = true, \\ may(v, r, n), & \text{wenn } r = Mem, \\ must(v, r, n), & \text{wenn } guard(n) = always \\ & \quad \wedge\, r \in dst(n) \cup Contains(n), \\ may(v, r, n), & \text{wenn } guard(n) = conditional \\ & \quad \vee\, (r \in PartOf(n) \wedge guard(n) \neq never), \\ v(r), & \text{sonst.} \end{cases}$$

Führt man mit dieser veränderten Transfer-Funktion die Datenflussanalyse durch, so würde man ohne die Speicheranalyse mit dem Slicing-Algorithmus aus Kapitel 5.6 den in Listing 6.1(c) dargestellten konservativen Slice berechnen. Für den gerade vorgestellten, erweiterten Slicing-Algorithmus dagegen, muss noch zusätzlich die Definition der Speicherzuordnungsfunktion erweitert werden. Diese ergibt sich wie folgt:

$$Memdep(n) = \begin{cases} Datadep(n, Mem), & \text{wenn } GetDefs(dfi_{mem}(n), a, b) = \mp \\ & \quad \wedge\, addr(n) = (a, b), \\ GetDefs(dfi_{mem}(n), a, b), & \text{wenn } Mem \in src(n) \\ & \quad \wedge\, addr(n) = (a, b), \\ \emptyset, & \text{sonst.} \end{cases}$$

Liefert die Speicheranalyse exakte Ergebnisse, werden diese zur Berechnung des Slices verwendet. Ist das Ergebnis ungenau, wird das Ergebnis der Datenabhängigkeitsanalyse verwendet. Für die Ressource Speicher liefert diese Analyse immer eine Überschätzung der tatsächlich für ein Intervall durchgeführten Speicherzugriffe. Somit ist garantiert, dass der erweiterte Slicing-Algorithmus aus Abschnitt 6.4 auch für diese Optimierung korrekte, wenn auch konservativere Slices berechnet.

Kapitel 7

Implementierung und Testergebnisse

Nachdem in den vorherigen Kapiteln zunächst die Grundlagen der Datenflussanalyse und der Abstrakten Interpretation vorgestellt und darauf aufbauend Datenflussanalysen zur Berechnung der Daten- und Kontrollabhängigkeiten eines Eingabeprogramms sowie ein dynamisches Speichermodell vorgestellt wurden, soll in diesem Kapitel die Implementierung der wichtigsten Komponenten und anschließend anhand von Beispielmessungen die Nutzbarkeit des in dieser Arbeit gewählten Ansatzes gezeigt werden.

7.1 Implementierung

Die Implementierung umfasst dabei neben den vorgestellten Datenflussanalysen und dem erweiterten Slicing-Algorithmus auch eine Kommunikationsschnittstelle, mit deren Hilfe die berechneten Slices in aiSee3 ([aiS]) graphisch dargestellt werden können. Darüberhinaus wurde eine Schnittstelle zum Einbringen von prozessorspezifischen Informationen entwickelt. Diese Teile stellen den generischen Teil des entwickelten Slicing-Algorithmus dar. Die vier Analysen wurden mittels des Werkzeugs PAG spezifiziert. Dieser Teil umfasst rund 1000 Zeilen Code. Die Implementierung der restlichen Teile erfolgte in C und umfasst rund 5000 Zeilen Quellcode.

Um die praktische Nutzbarkeit der vorgestellten Analysen und des Algorithmus zu zeigen, wurde als Beispielarchitektur die ARM9 Architektur ([ARM00]) ausgewählt. Die prozessorspezifischen Anpassungen des Slicing-Algorithmus für diese Architektur umfassen ca. 200 Zeilen Code, wovon die Hälfte der Verfeinerung der Ergebnisse durch Hinzufügen von semantischen Informationen (vgl. 5.7.2.2) dient. Der Slicing-Algorithmus selbst wurde analog zu Listing 6.3 implementiert, seine Terminierung ist somit garantiert.

Die folgenden Abschnitte zeigen zunächst einige Anpassungen von Definitionen, die für die Verwendung von PAG durchgeführt werden mussten. Danach wird die Spezifikation einer Datenflussanalyse mit PAG vorgestellt. Dazu werden die Spezifikationssprachen *DATLA* und *FULA* kurz vorgestellten. Um die Anpassung des Slicing-

Werkzeugs an andere Architekturen so einfach wie möglich zu gestalten wird dann eine Schnittstelle zum Einbringen von prozessorspezifischen Informationen vorgestellt.

7.1.1 Vorbetrachtungen

In Kapitel 5 wurde eine abstrakte Semantik vorgestellt, mit deren Hilfe statisch die Programmabhängigkeiten von disassemblierten Eingabeprogrammen bestimmt und zu beliebigen Kriterien Slices berechnet werden können. Dazu wurden verschiedene Analysen vorgestellt, die auf dieser abstrakten Semantik durchgeführt werden können. Zur Realisierung dieser Analysen wird der in Abschnitt 4.3 vorgestellte Programm Analysator Generator verwendet. Zur Verarbeitung der Zwischendarstellung stellt PAG ein Frontend zur Verfügung, das aus der CRL-Beschreibung eine den Kontrollflussgraphen beschreibende Datenstruktur generiert und eine Schnittstelle zur Durchführung von Analysen anbietet.

Wie bereits bei der Vorstellung von PAG näher beschrieben, wird dabei für interprozedurale Analysen ein statischer Call Graph mit Basisblock-Optimierung verwendet. D.h. die Datenflussanalyse wird auf zusammenhängenden Blöcken durchgeführt, die berechneten Datenflusswerte werden dabei nur am Eingang und Ausgang eines jeden Basisblocks gespeichert und für die einzelnen Instruktionen bei Bedarf neu berechnet. Dies dient vor allem dazu, den Speicherverbrauch der Analysen in einem moderaten Rahmen zu halten. Zudem wird jedem Basisblock ein Feld zugeordnet, dessen einzelne Elemente verschiedene Kontexte des betreffenden Knotens darstellen. Somit lassen sich, wie bereits in Abschnitt 4.3.1 dargestellt, verschiedene Aufrufe ein und derselben Prozedur unterscheiden.

Dieses Konzept führt dazu, dass ein Knoten eines Kontrollflussgraphen eindeutig durch ein Tupel $(b, i, c) \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$ beschrieben werden kann. Dabei gibt b die Nummer des Basisblocks innerhalb des Kontrollflussgraphen, i die Nummer der Instruktion innerhalb des Basisblocks b und c mit $0 \leq c < arr(b, i)$ den Kontext des betreffenden Knotens an.

Diese Erweiterung muss nachfolgend in alle bisherigen Definitionen übernommen werden. Im folgenden wird nun beschrieben, wie sich die vorgestellten Analysen in PAG realisieren lassen.

7.1.2 Spezifikation einer Datenflussanalyse

Die Spezifikation einer Datenflussanalyse in PAG besteht aus zwei Teilen: Einem Teil, in dem die für die Analyse benötigten Datentypen und Verbände spezifiziert werden und einem Teil zur Beschreibung der Transferfunktionen und des eigentlichen Datenflussproblems. Dazu stellt PAG zwei eigene Sprachen, *DATLA* und *FULA*, zur

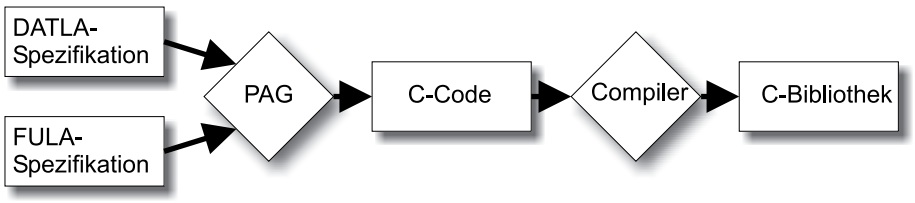


Abbildung 7.1: Generierung von Datenflussanalysatoren mit PAG

Verfügung. Aus dieser Spezifikation einer Analyse wird dann C-Code generiert, der alle Funktionen zur Durchführung der beschriebenen Datenflussanalyse enthält. Der generierte Code ist effizient ([AMW95]) und kann problemlos in bestehenden Projekten verwendet werden. Die Generierung eines Analysators ist in Abbildung 7.1 graphisch dargestellt.

In den folgenden beiden Abschnitten soll anhand der in Abschnitt 5.4 vorgestellten Datenabhängigkeitsanalyse die PAG-Spezifikation beschrieben werden.

7.1.2.1 Verbandsspezifikation

```
SET
    tup     = snum * snum * snum
DOMAIN
    tupset  = set( tup )
    map     = snum -> tupset
    func    = lift( map )
```

Listing 7.1: DATLA-Spezifikation der Datenabhängigkeitsanalyse

Dieser Abschnitt beschreibt, wie man mittels der Sprache *DATLA* Datentypen und Verbände in PAG spezifizieren kann. Dazu dienen die beiden Schlüsselwörter `SET` und `DOMAIN`. Dabei können nur Datentypen bzw. Verbände als Grundlage für eine Datenflussanalyse dienen, die nach dem Schlüsselwort `DOMAIN` spezifiziert wurden.

Die zur Datenabhängigkeitsanalyse benötigten Datentypen und Verbände sind in Listing 7.1 dargestellt. Aufbauend auf dem vordefinierten Datentyp `snum`, der die ganzen Zahlen repräsentiert, wird zunächst der Datentyp `tup` generiert. Dieser Datentyp repräsentiert die in Abschnitt 7.1.1 vorgestellte erweiterte Definition eines Programmpunktes. Auf diesem Datentyp aufbauend wird die Menge `tupset`, die die Potenzmenge der Programmpunkte mit der partiellen Ordnung $\subseteq$ darstellt, generiert. Daraus wird dann mittels des `->`-Operators eine Funktion und anschließend der um ein zusätzliches größtes und kleinstes Element erweiterte Verband `func` gebildet. Dieser Verband repräsentiert den in Kapitel 5.4 vorgestellten Wertebereich für die Datenabhängigkeitsanalyse. Weitere Einzelheiten zu *DATLA* finden sich in [Abs02].

7.1.2.2 Problemspezifikation

Nachdem der Verband der gewünschten Analyse spezifiziert ist, kann dieser in der Beschreibung des Datenflussproblems verwendet werden. Dazu unterscheidet PAG drei verschiedene Sektionen: die Problembeschreibung, die Beschreibung der Transferfunktionen und einen Teil zur Beschreibung von Hilfsfunktionen.

```
PROBLEM       verfügbare Definitionen
direction   : forward
carrier     : func
init        : bot
init_start  : lift( bot )
combine     : comb
widening    : wid
equal       : eq
retfunc     : ret
```

Listing 7.2: Problembeschreibung der Datenabhängigkeitsanalyse

Listing 7.2 zeigt die Problembeschreibung der Datenabhängigkeitsanalyse. Dieser Teil wird mit dem Schlüsselwort `PROBLEM` eingeleitet und umfasst neben der Richtung des Problems (vorwärts oder rückwärts) auch den Verband, auf dem die Analyse durchgeführt werden soll. Daneben gibt man an, mit welchen Werten des Verbandes die Fixpunktiteration gestartet werden soll (`init`). Der initiale Wert des Startpunkts der Analyse muss einzeln angegeben werden (`init_start`). Danach definiert man noch, wie der Datenflusswert bei zusammenfließenden Kontrollfluss berechnet werden soll (`combine`) und welche Funktion den Widening-Operator implementiert (`widening`). Zusätzlich gibt man noch eine Funktion an, mit der der Stabilitätstest der Fixpunktiteration durchgeführt werden soll (`equal`). Die Return-Funktion wird nach dem Schlüsselwort `retfunc` eingeführt. Die Implementierung der angegebenen Funktionen erfolgt dann in der `SUPPORT`-Sektion (vgl. Listing 7.4).

```
TRANSFER
normal_node(), crl_false:
  //die betreffende Instruktion wird über diesen Pfad nicht
     ausgeführt
  @;

normal_node(), _:
  //die Standardregel für Instruktionen
  let
    dst=get_destinations( [dst1,dst2,dst3,dst4,
                           dst5,dst6,dst7,dst8]);
```

```
  in
12    let
        must=get_must_resources( dst );
14    in
        let
16        may=get_may_resources( must );
        in
18        update( update( @,must,block,instruction,context,guard ),
                  may,
20                block,
                  instruction,
22                context,
                  0 );
24
_, _:
26  //keine Veränderung am Datenflusswert notwendig
  @;
```

Listing 7.3: Transfer-Funktion der Datenabhängigkeitsanalyse

Listing 7.3 zeigt die Beschreibung der Transferfunktion der Datenabhängigkeitsanalyse. Dieser Teil der Spezifikation wird mit dem Schlüsselwort `TRANSFER` begonnen. Zur Beschreibung der Funktionen stellt PAG die funktionale Sprache *FULA* zur Verfügung. Sie erlaubt es, die Transferfunktion aus mehreren einzelnen Regeln oder Mustern zu beschreiben. Ein Muster besteht immer aus einem Knotentyp und einem Kantentyp. Damit lässt sich jeder Knoten, bzw. jeder mögliche Pfad in einem Supergraphen eindeutig einer Regel zuordnen. Alternativ kann auch der Wildcard-Operator _ in einem Muster verwendet werden. CRL unterscheidet folgende Knotentypen:

- normal_node: alle Instruktionen
- call_node: Aufrufknoten
- return_node: der Rückkehrknoten nach einem Aufrufknoten
- entry_node: der Startknoten einer Prozedur
- exit_node: den Endknoten einer Prozedur

Das CRL-Frontend unterscheidet Kantentypen:

- bb_intern: eine Kontrollflusskante zwischen Instruktionen eines Basisblocks
- crl_local: Kante zwischen Call- und dem zusammengehörigen Return-Knoten

- crl_true
- crl_false
- crl_call: Kante zwischen Call- und Startknoten
- crl_return: Kante zwischen Exit- und Return-Knoten

Mit Hilfe dieser Muster lässt sich die Transfer-Funktion einer Datenflussanalyse, die auf der abstrakten Semantik CRL durchgeführt wird, beschreiben.

Alle Funktionen, die innerhalb der `TRANSFER` und der `PROBLEM`-Sektion aufgerufen werden, um so den neuen Datenflusswert zu berechnen, müssen in der `SUPPORT`-Sektion eingeführt werden.

```
SUPPORT

comb( a, b )       = a lub b;
wid( old, new )    = new;
eq( a, b )         = a = b;
ret( top, _ )      = top;
ret( _, top )      = top;
ret( call, bot )   = call;
ret( bot, exit )   = exit;
ret( call, exit ) = lift( set_new_values( drop( call ),
                                          drop( exit ),
                                          CallerSaved()));

set_new_values :: map, map, snuml -> map;
set_new_values( f, _, [] ) = f;
set_new_values( f, g, reg:regs ) =
                  set_new_values( f\[x->g{!reg!}], g, regs );

update :: func, snuml, snum, snum, snum, bool -> func;
update ( top, _, _, _, _, _ ) = top;
update ( bot, _, _, _, _, _ ) = bot;
update ( ff, list, b, i, c, g ) =
    if ( g = true )
    then
      lift( must( drop( f ), list, b, i, c ))
    else
      lift( may( drop( f ), list, b, i, c ))
    endif;

must :: map, snuml, snum, snum, snum -> map;
```

```
must( f, [], _, _, _ ) = f;
must( f, reg:regs, b, i, c ) =
    must(lift(f\[reg->((b,i),c)]),regs, b, i, c );

may :: map, snuml, snum, snum, snum -> map;
may( f, [], _, _, _ ) = f;
may( f, reg:regs, b, i, c ) =
  may(lift(f\[reg->f{!reg!}~((b,i),c)]), regs, b, i, c );

get_destinations :: strl  -> snuml;
get_must_updates :: snuml -> snuml;
get_may_updates  :: snuml -> snuml;
```

Listing 7.4: Hilfsfunktionen der Datenabhängigkeitsanalyse

Listing 7.4 zeigt die Beschreibung der Hilfsfunktionen, die für die Datenabhängigkeitsanalyse benötigt werden. Diese Funktionen können sowohl in der funktionalen Programmiersprache *FULA* oder in C definiert werden. Im obigen Listing sind beispielsweise die Funktionen `update()`, `must`, `may`, `ret()` und `set_new_value()` in *FULA* definiert, während für die Funktionen `get_destinations()`, `get_may_updates()` und `get_must_updates()` nur Prototypen definiert wurden. Die Implementierung dieser Funktionen erfolgt in C-Code direkt und wird in Abschnitt 7.1.3 beschrieben. Die Funktion `lub` ist eine von PAG vordefinierte Funktion, die zu zwei gegebenen Verbandselementen die Vereinigung $\bigsqcup$ dieser berechnet. Eine komplette Beschreibung der funktionalen Sprache *FULA* findet sich in [Abs02], eine ausführliche Beschreibung des CRL-Frontends findet sich in [Abs].

Mit Hilfe dieser beiden Teile lassen sich mit PAG effiziente Datenflussanalysatoren generieren. Die so erzeugten C-Dateien lassen sich mit einem C-Compiler in eine Bibliothek übersetzen und dann in eigenen Projekten verwenden. Alle in dieser Arbeit vorgestellten Analysen sind auf diese Weise implementiert.

7.1.3 Schnittstelle für prozessorspezifische Eigenschaften

Dieser Abschnitt beschreibt die Schnittstelle, mit deren Hilfe sich prozessorspezifische Informationen über die Struktur der Register sowie Aufruf-Konventionen zu den Analysen hinzufügen lassen.

Den Kern der Implementierung stellt dabei die Realisierung der hierarchischen Ordnung für zusammengesetzte Register dar (vgl. 5.4.1). Diese werden in Form eines Baumes realisiert.

Diese Struktur muss für jede Prozessor-Architektur einzeln aufgebaut werden. Dazu dient die Funktion:

```
register_alias_add(int number_of_childs, char* parent, char* child1, ...);
```

Zum Aufbauen der Struktur beginnt man mit dem größten Register und gibt die Namen der Kindregister mit an. Werden mehr als zwei Hierarchieebenen benötigt, wiederholt man diesen Vorgang mehrfach.

Zum Abfragen der Struktur dienen die beiden Funktionen:

```
register_alias_get_childs ( char* reg );
```

und

```
register_alias_get_parents ( char* reg );
```

Diese liefern für ein gegebenes Register alle Kinder bzw. alle Väter zurück (Typ `snuml`). Diese Informationen werden dann in der Datenabhängigkeitsanalyse verwendet.

Darüberhinaus existiert eine Funktion

```
CallerSaved( void );
```

die eine Liste von Registern zurückliefert, die von einer Prozedur vor einem Call auf dem Stack gespeichert werden und bei der Rückkehr wieder geladen werden. Diese Funktion realisiert die in Abschnitt 5.7.2.1 beschriebenen Aufruf-Konventionen von Prozessoren.

Die oben vorgestellten Funktionen müssen für neue Architekuren angepasst werden. Der Aufwand dazu lässt sich als recht gering einstufen. Die Definition von Aufruf-Konventionen ist darüberhinaus nicht unbedingt erforderlich - eine Definition verfeinert allerdings, analog zur Angabe von semantischen Informationen, die Analysen, was in kleineren Slices resultiert.

7.2 Testergebnisse

Dieser Abschnitt soll einen Überblick über die praktische Nutzbarkeit des entwickelten Slicing-Werkzeuges bieten. Dazu wurden verschiedene Testprogramme ausgewählt, die so in der Industrie unter anderem zur Qualitätssicherung von Softwareprodukten eingesetzt werden.

Die Nutzbarkeit des entwickelten Slicing-Werkzeuges hängt dabei ab von:

- Zeit, die benötigt wird, die CRL-Beschreibung in eine Datenstruktur zu überführen.

Programm	minmax	fac	prime	dry2_1	st30
Routinen	4	2	4	17	163
Instruktionen	114	24	119	773	3820
Calls	5	2	4	32	309
Schleifen	0	1	2	9	50
Loads	4	2	20	296	984
Stores	4	2	10	140	877

Tabelle 7.1: Charakteristika der Testprogramme

- Zeit zur Durchführung der beschriebenen Analysen.
- Zeit zur Berechnung eines Slices für beliebige Kriterien.

Leider lässt sich zum letzten Punkt keine allgemein gültigen Aussage über die Laufzeit treffen. Die Berechnungszeit eines Slices hängt dazu zu sehr vom ausgewählten Slicing-Kriterium ab. Die Komplexität des vorgestellten Algorithmus wurde aber bereits in Abschnitt 6.4 vorgestellt. In allen Beispielmessungen lag die Berechnungszeit für die Berechnung eines Slices für beliebige Kriterien immer deutlich unter einer Sekunde. Diese Zeit hängt aber stark von der Größe des analysierten Eingabeprogramms ab.

Interessant für eine praktische Nutzbarkeit sind aber auch die beiden ersten Zeitkomponenten. Diese können unter dem Begriff Vorberechnungszeit zusammengefasst werden. Die Erzeugung der CRL-Zwischendarstellung lässt sich durch verschiedene Annotationen beeinflussen. Die Eingabedateien für den Slicing-Algorithmus sind danach unabhängig von Benutzereingaben jeglicher Art. Somit lassen sich dazu Aussagen treffen.

Als Testprogramme wurden Programme ausgewählt, die in der Industrie teilweise zur Qualitätssicherung eingesetzt werden. Darüberhinaus wurde ein Programm analysiert, das in dieser Form in Kraftfahrzeugen der neuesten Generation eingesetzt wird. Die Programme wurden wegen ihren unterschiedlichen Eigenschaften ausgewählt: Während `minmax` ein Programm ist, bei dem weder Schleifen noch Rekursion den Kontrollfluss schwer analysierbar machen, so ist `fac` die rekursive Implementierung der Fakultätsfunktion. Diese wird für mehrere Eingaben innerhalb einer Schleife aufgerufen. Das Programm `prime` dient zum Testen von Schleifen alleine. Als echtes Benchmark-Programm wurde zudem noch `dry2_1` analysiert, das eine Umsetzung von *drystone* ist. Um die Verwendbarkeit auch in großen Softwaresystemen zu testen, dient das Programm `st30`, das zur Reifendrucküberwachung moderner Kraftfahrzeuge eingesetzt wird.

Tabelle 7.1 zeigt die charakteristischen Eigenschaften der Testprogramme. Der Assemblercode der Beispielprogramme `minmax`, `fac` und `prime` findet sich im Anhang

Programm	KFG	D_{rd}	D_{pdom}	D_{cp}	D_{mem}	$\sum$
minmax	0,03 s	0,02 s	$<$ 0,01 s	$<$ 0,01 s	$<$ 0,01 s	0,07 s
fac	0,02 s	0,03 s	$<$ 0,01 s	0,02 s	0,02 s	0,10 s
prime	0,03 s	0,04 s	$<$ 0,01 s	0,02 s	0,02 s	0,13 s
dry2_1	0,17 s	0,25 s	0,07 s	0,10 s	0,32 s	0,82 s
st30	0,50 s	173,20 s	0,40 s	0,40 s	1,20 s	177,50 s

Tabelle 7.2: Vorberechnungszeiten für die Beispielprogramme

dieser Arbeit. Die Tabelle zeigt die für die Laufzeit der Analysen relevanten Eigenschaften: die Anzahl der Routinen, die Anzahl der Instruktionen, die Anzahl der Call-Stellen sowie die Anzahl der Schleifen. Die Geschwindigkeit der Analysen hängt dabei von der Anzahl der Iterationen der Fixpunktberechnung ab. Für die Speicheranalyse ist zudem die Anzahl der Load-/Store-Instruktionen interessant.

Die Messung der Zeiten wurde auf einem Rechner mit Pentium III 1200 MHz CPU und 256 MB SD-RAM durchgeführt. Als Betriebssystem diente eine Debian Woody Installation. Alle Messungen wurden mehrfach durchgeführt und der Mittelwert der gemessenen Zeiten bestimmt. Die Messgenauigkeit lag bei 0,01 sek. Die Ergebnisse der Messungen für die verschiedenen Testprogramme sind in Tabelle 7.2 dargestellt. In den verschiedenen Spalten sind die Zeiten aufgelistet, die benötigt werden, um die CRL-Datei einzulesen und den Kontrollflussgraphen aufzubauen (KFG), sowie die Zeiten zur Lösung der vier Datenflussprobleme (D_{rd}, D_{pdom}, D_{cp} und D_{mem}). Die letzte Spalte zeigt die Summe der vorher angegeben Zeiten. Diese kann, wie zuvor erwähnt, als Vorberechnungszeit angesehen werden.

Eine weitere wichtige Komponente für die praktische Nutzbarkeit des Werkzeuges ist die Qualität des berechneten Slices. Qualität meint dabei, um wieviel Prozent der berechnete Slice S von einem minimalen Slice S_M für das gleiche Kriterium abweicht. Diese Abweichung A eines Slices S lässt sich also als Masszahl wie folgt definieren:

$$A_S = \left(\frac{|S|}{|S_M|} - 1 \right) * 100$$

Da für ein Slicing-Kriterium der minimale Slice nicht notwendigerweise eindeutig ist und das Problem der Bestimmung von minimalen Slices nicht entscheidbar ist (vgl. [Wei82], [Wei84] und [Wei79]), kann man keine allgemein gültige Aussagen treffen. Daher soll hier anhand einiger Eingabeprogramme für ausgewählte Kriterien die Qualität der berechneten Slices bestimmt werden.

Dazu werden die beiden Programme `minmax` und `fac` verwendet. Die Abbildungen 7.2, 7.3 und 7.4 zeigen die Kontrollflussgraphen der beiden Programme. Für jedes der beiden Programme wurde anhand von drei verschiedenen Slicing-Kriterien von Hand jeweils ein minimaler Slice ermittelt. Danach wurden für die gleichen Kriterien Slices

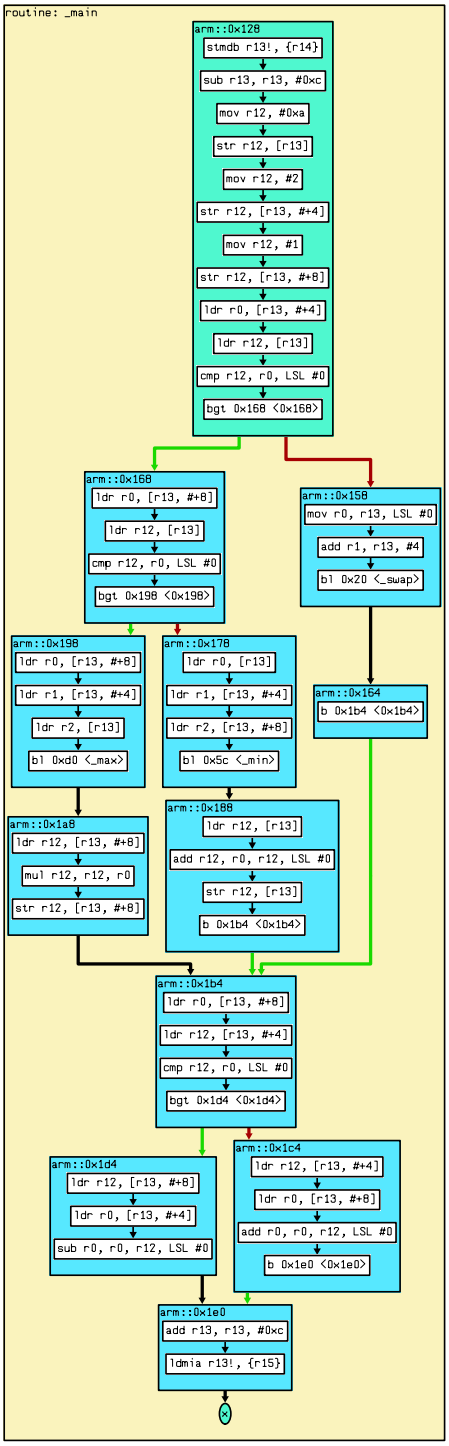


Abbildung 7.2: Kontrollflussgraph von `minmax`

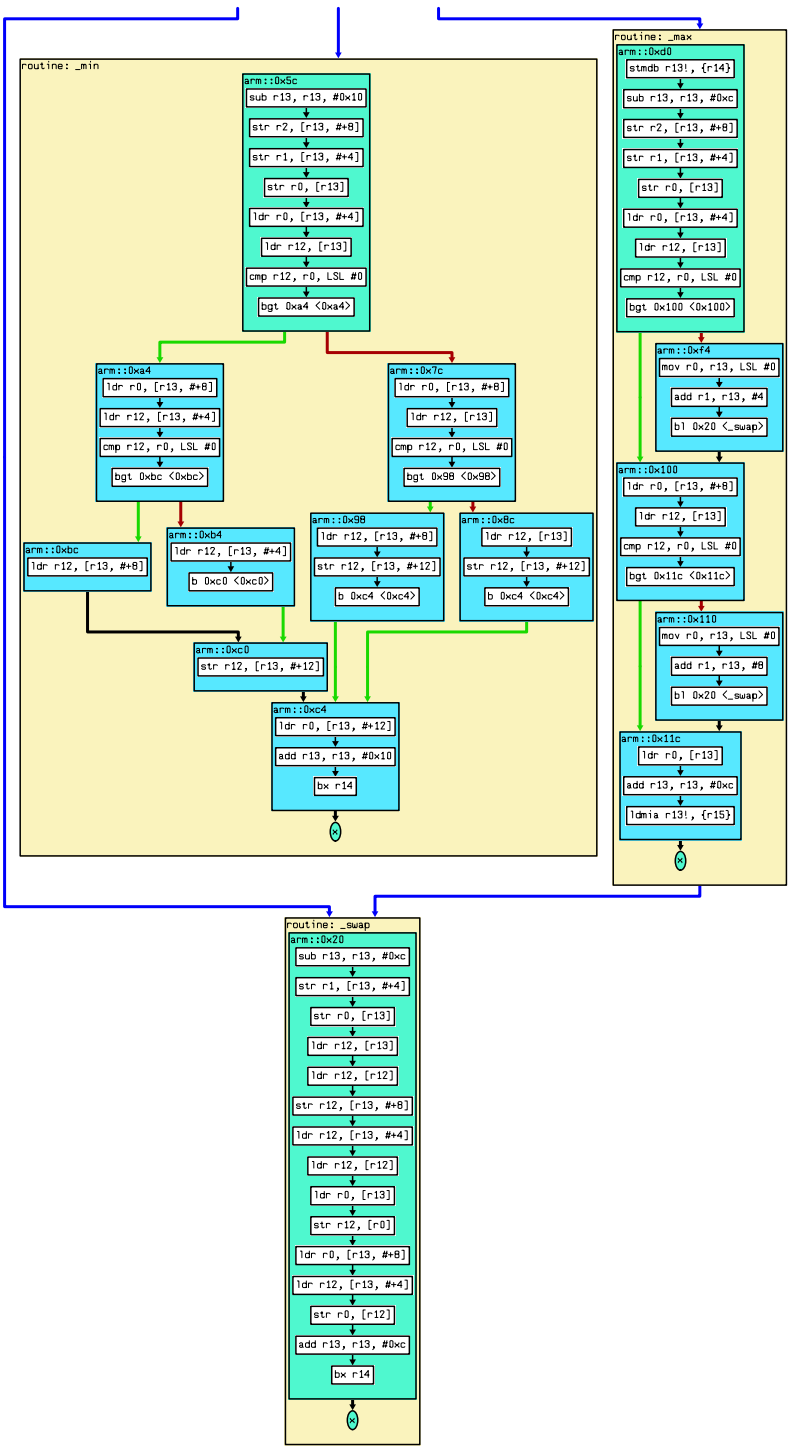


Abbildung 7.3: Kontrollflussgraph von `minmax` (Fortsetzung)

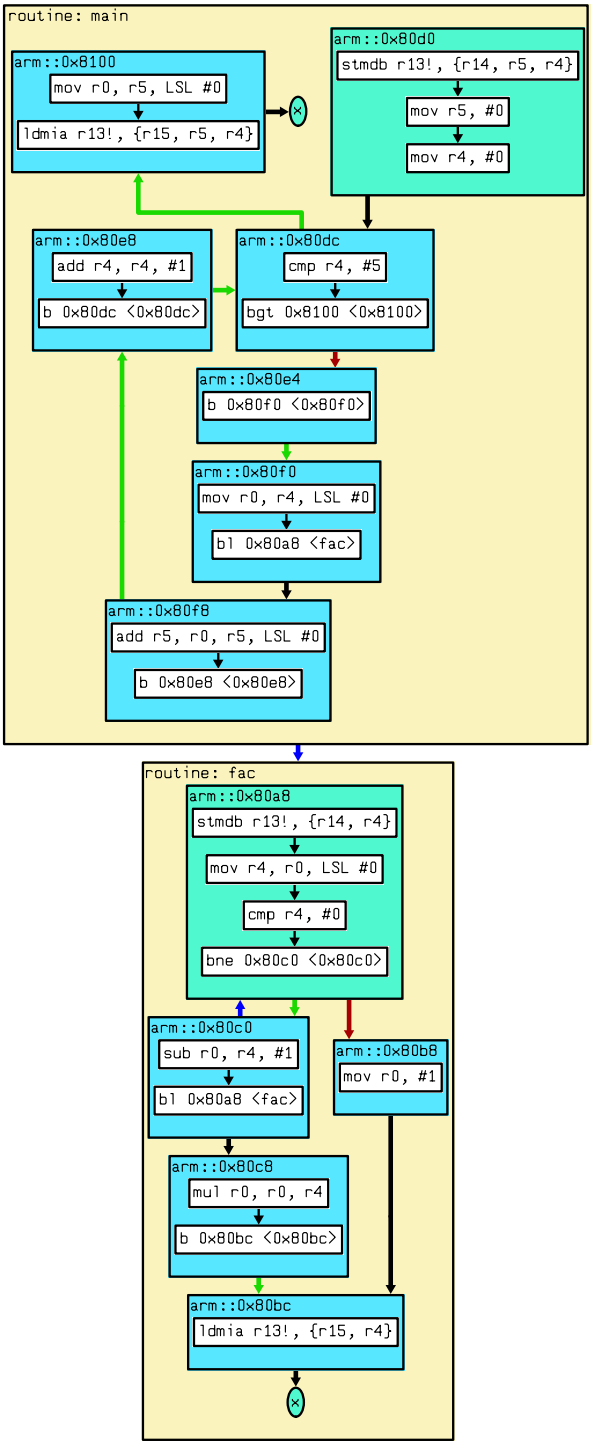


Abbildung 7.4: Kontrollflussgraph von `fac`

Kriterium	$(0x154, CPSR)$	$(0x124, r13)$	$(0x40, r0)$
Kontext	-	-	1. Aufruf aus `_max`
Instruktionen			
minimaler Slice S_M	8	6	19
berechneter Slice S	10	6	20
Abweichung A_S	25%	0%	5,2%

Tabelle 7.3: Abweichungen der Slices für `minmax`

Kriterium	$(0x80f0, r4)$	$(0x80c0, r0)$	$(0x80c0, r0)$
Kontext	-	1. Aufruf	1. rekursiver Aufruf
Instruktionen			
minimaler Slice S_M	5	7	9
berechneter Slice S	5	7	9
Abweichung A_S	0%	0%	0%

Tabelle 7.4: Abweichungen der Slices für `fac`

mittels der durch die vorgestellten Analysen berechneten Abhängigkeiten berechnet. Dabei lag die Zeit zur Berechnung der unterschiedlichen Slices immer deutlich unter 1 Sekunde. Die Ergebnisse sind in den Tabellen 7.3 und 7.4 dargestellt. Die Slicing-Kriterien, die beispielhaft ausgewählt wurden, sind in der ersten Zeile dargestellt. Die Anzahl der Instruktionen des minimalen bzw. des berechneten Slice für das entsprechende Kriterium sind in der Tabelle angegeben. Die prozentuale Abweichung findet sich in der letzten Zeile.

Betrachtet man die Vorberechnungszeit und die Qualität der berechneten Slices zusammen, so zeigt sich, dass der gewählte Ansatz mit minimalem prozessorspezifischem Wissen teilweise präzise Slices für die gewählten Kriterien liefert. Auch die Dauer der Vorberechnungen bewegt sich in einem Rahmen, der einen Einsatz innerhalb von Softwareentwicklungsumgebungen durchaus zulässt.

Kapitel 8

Zusammenfassung und Ausblick

Es wurde in dieser Arbeit gezeigt, wie man mit Hilfe von Datenflussanalysen und der Abstrakten Interpretation die Daten- und Kontrollabhängigkeiten von disassemblierten Programmen berechnen kann. Die Kenntnis der Instruktionssemantik der zugrundeliegenden Rechnerarchitektur ist bei dem vorgestellten Verfahren nur zur Erzeugung einer geeigneten Zwischendarstellung notwendig. Lediglich zur Berechnung der Datenabhängigkeiten auf Basis eines Kontrollflussgraphen musste rechnerspezifisches Wissen in die Analyse eingebracht werden. Darüberhinaus wurde ein dynamisches Modell zur effizienten Modellierung von Speicherzugriffen vorgestellt. Aufbauend auf den Ergebnissen einer externen Werte-Analyse wurde ein Datenflussproblem formuliert, mit dem es möglich ist, Speicherzugriffe innerhalb des zu analysierenden Programms zu unterscheiden.

Die Ergebnisse dieser Analysen wurden dazu verwendet, einen effizienten Slicing-Algorithmus zu entwickeln, der mit minimalem hardwarespezifisches Wissen Slices auf der Disassemblerebene berechnen kann. Zusätzlich wurden einige Optimierungen vorgestellt, mit denen sich die Analysen weiter beschleunigen oder präzisieren lassen.

Die verschiedenen Analysen und der Slicing Algorithmus wurden zu einem Werkzeug vereint, mit dem verschiedene Tests durchgeführt wurden. Diese Tests zeigen zum einen die Effizienz des gewählten Ansatzes, zum anderen aber auch die Qualität der berechneten Slices. Dazu wurde anhand einiger Beispiele gezeigt, dass die mit dem vorgestellten Algorithmus berechneten Slices nur wenig von den minimalen Slices für ein gewähltes Kriterium abweichen.

Um den Slicing-Algorithmus, sowie die Analysen so generisch wie möglich zu halten, wurde zudem eine Schnittstelle für das Einbringen von semantischen Informationen zur Verfeinerung der Analysen entwickelt. Das dadurch eingebrachte architekturspezifische Wissen kann zur exakteren Berechnung der Abhängigkeiten verwendet werden und liefert dadurch qualitativ hochwertigere Slices.

Soweit mir dies bekannt ist, stellt der beschriebene Ansatz den bisher einzigen generischen Slicing Algorithmus für die Disassemblerebene dar. Die Modellierung von Speicherzugriffen unter Verwendung der Ergebnisse einer externen Werte-Analyse ist,

wie bereits erwähnt, effizient und in dieser Form auf verschiedene andere Anwendungen übertragbar.

Das im Rahmen dieser Diplomarbeit entwickelte Werkzeug wurde mittlerweile mit Erfolg in ein Framework zur Laufzeitbestimmung von ausführbaren Programmen integriert. Dort kann es unter anderem als Hilfe zur Annotation von nicht automatisch auflösbaren Sprüngen und Aufrufen eingesetzt werden. Dazu wurde, wie bereits zuvor erwähnt, eine Kommunikationsschnittstelle entwickelt, die es dem Benutzer erlaubt, Slicing-Kriterien interaktiv aus einem Graph-Betrachtungs-Programm heraus auszuwählen. Die berechneten Slices bezüglich des so gewählten Kriteriums können dann ebenfalls in dem Graph-Betrachtungs-Programm visualisiert werden.

Für die Zukunft ist bereits eine Erweiterung des bisher nur auf Rückwärts-Slicing beschränkten Werkzeugs auf Vorwärts-Slices geplant. Wie in Kapitel 2 angedeutet, können dazu die gleichen Methoden und Techniken verwendet werden, die auch für Rückwärts-Slices verwendet wurden. Lediglich die Richtung der Analysen muss dazu umgekehrt werden und die Betrachtung der Datenabhängigkeiten, die sich bisher nur auf die Flussabhängigkeit beschränkte, muss auf Benutzungs-Setzungs-Abhängigkeit erweitert werden.

Anhang A

Testprogramme

Die folgenden Seiten enthalten eine Auflistung der Beispielprogramme. Der dargestellte Assemblercode wurde von Compilern für ARM-Architekturen erzeugt. Dazu wurden Compiler von Texas Instruments und ARM verwendet. Die Bedeutung der einzelnen Instruktionen findet sich in [ARM00] und [ARM99]. Auf die Darstellung von Architektur-spezifischen Programmfragmenten, wie z.B. dem Code zum Wechsel vom 16-bit Thumb Modus zum 32-bit ARM Modus, sowie auf Assemblerdirektiven wurde hier verzichtet. Eine Beschreibung dieser findet sich [ARM01].

A.1 minmax

```
.routine _main
  stmdb     r13!, {r14}
  sub       r13, r13, #0xc
  mov       r12, #0xa
  str       r12, [r13]
  mov       r12, #2
  str       r12, [r13, #+4]
  mov       r12, #1
  str       r12, [r13, #+8]
  ldr       r0, [r13, #+4]
  ldr       r12, [r13]
  cmp       r12, r0, LSL #0
  bgt       L1
  mov       r0, r13, LSL #0
  add       r1, r13, #4
  bl        _swap
  b         L3
L1:
  ldr       r0, [r13, #+8]
  ldr       r12, [r13]
  cmp       r12, r0, LSL #0
  bgt       L2
  ldr       r0, [r13]
  ldr       r1, [r13, #+4]
  ldr       r2, [r13, #+8]
  bl        _min
  ldr       r12, [r13]
  add       r12, r0, r12, LSL #0
  str       r12, [r13]
  b         L3
L2:
  ldr       r0, [r13, #+8]
  ldr       r1, [r13, #+4]
  ldr       r2, [r13]
  bl        _max
  ldr       r12, [r13, #+8]
  mul       r12, r12, r0
  str       r12, [r13, #+8]
L3:
  ldr       r0, [r13, #+8]
  ldr       r12, [r13, #+4]
  cmp       r12, r0, LSL #0
  bgt       L4
  ldr       r12, [r13, #+4]
  ldr       r0, [r13, #+8]
  add       r0, r0, r12, LSL #0
  b         L5
L4:
  ldr       r12, [r13, #+8]
  ldr       r0, [r13, #+4]
  sub       r0, r0, r12, LSL #0
L5:
  add       r13, r13, #0xc
  ldmia     r13!, {r15}
.end

.routine _max
  stmdb     r13!, {r14}
  sub       r13, r13, #0xc
  str       r2, [r13, #+8]
  str       r1, [r13, #+4]
  str       r0, [r13]
  ldr       r0, [r13, #+4]
  ldr       r12, [r13]
  cmp       r12, r0, LSL #0
  bgt       L6
  mov       r0, r13, LSL #0
  add       r1, r13, #4
  bl        _swap
L6:
  ldr       r0, [r13, #+8]
  ldr       r12, [r13]
  cmp       r12, r0, LSL #0
  bgt       L7
  mov       r0, r13, LSL #0
  add       r1, r13, #8
  bl        _swap
L7:
  ldr       r0, [r13]
  add       r13, r13, #0xc
  ldmia     r13!, {r15}
.end

.routine _min
  sub       r13, r13, #0x10
  str       r2, [r13, #+8]
  str       r1, [r13, #+4]
  str       r0, [r13]
```

Listing A.1: Assemblercode des Testprogramms `minmax`

```
    ldr     r0, [r13, #+4]
    ldr     r12, [r13]
    cmp     r12, r0, LSL #0
    bgt     L8
    ldr     r0, [r13, #+8]
    ldr     r12, [r13]
    cmp     r12, r0, LSL #0
    bgt     L9
    ldr     r12, [r13]
    str     r12, [r13, #+12]
    b       L12
L9:
    ldr     r12, [r13, #+8]
    str     r12, [r13, #+12]
    b       L12
L8:
    ldr     r0, [r13, #+8]
    ldr     r12, [r13, #+4]
    cmp     r12, r0, LSL #0
    bgt     L10
    ldr     r12, [r13, #+4]
    b       L11
L10:
    ldr     r12, [r13, #+8]
L11:
    str     r12, [r13, #+12]
L12:
    ldr     r0, [r13, #+12]
    add     r13, r13, #0x10
    bx      r14
.end
```

```
.routine _swap
    sub     r13, r13, #0xc
    str     r1, [r13, #+4]
    str     r0, [r13]
    ldr     r12, [r13]
    ldr     r12, [r12]
    str     r12, [r13, #+8]
    ldr     r12, [r13, #+4]
    ldr     r12, [r12]
    ldr     r0, [r13]
    str     r12, [r0]
    ldr     r0, [r13, #+8]
    ldr     r12, [r13, #+4]
    str     r0, [r12]
    add     r13, r13, #0xc
    bx      r14
.end
```

Listing A.2: Assemblercode des Testprogramms `minmax` (Fortsetzung)

A.2 fac

```
.routine main
  stmdb     r13!, {r14, r5, r4}
  mov       r5, #0
  mov       r4, #0
L1:
  cmp       r4, #5
  bgt       L2
  b         L3
L4:
  add       r4, r4, #1
  b         L1
L3:
  mov       r0, r4, LSL #0
  bl        fac
  add       r5, r0, r5, LSL #0
  b         L4
L2:
  mov       r0, r5, LSL #0
  ldmia     r13!, {r15, r5, r4}
.end

.routine fac
  stmdb     r13!, {r14, r4}
  mov       r4, r0, LSL #0
  cmp       r4, #0
  bne       L5
  mov       r0, #1
L6:
  ldmia     r13!, {r15, r4}
L5:
  sub       r0, r4, #1
  bl        fac
  mul       r0, r0, r4
  b         L6
.end
```

Listing A.3: Assemblercode des Testprogramms `fac`

A.3 prime

```
.routine _prime
  stmdb     r13!, {r14}
  sub       r13, r13, #8
  str       r0, [r13]
  ldr       r0, [r13]
  bl        _even
  cmp       r0, #0
  beq       L1
  mov       r0, #0
  ldr       r12, [r13]
  cmp       r12, #2
  bne       L2
  mov       r0, #1
  b         L2
L1:
  mov       r12, #3
  str       r12, [r13, #+4]
  ldr       r0, [r13, #+4]
  ldr       r12, [r13, #+4]
  mul       r12, r12, r0
  ldr       r0, [r13]
  cmp       r12, r0, LSL #0
  bhi       L3
L5:
  ldr       r0, [r13, #+4]
  ldr       r1, [r13]
  bl        _divides
  cmp       r0, #0
  beq       L4
  mov       r0, #0
  b         L2
L4:
  ldr       r12, [r13, #+4]
  add       r12, r12, #2
  str       r12, [r13, #+4]
  ldr       r0, [r13, #+4]
  ldr       r12, [r13, #+4]
  mul       r0, r0, r12
  ldr       r12, [r13]
  cmp       r0, r12, LSL #0
  bls       L5
L3:
  mov       r0, #0
  ldr       r12, [r13]
  cmp       r12, #1
  bls       L2
  mov       r0, #1
L2:
  add       r13, r13, #8
  ldmia     r13!, {r15}
.end

.routine _divides
  stmdb     r13!, {r14}
  sub       r13, r13, #8
  str       r1, [r13, #+4]
  str       r0, [r13]
  mov       r12, #0
  ldr       r0, [r13, #+4]
  ldr       r1, [r13]
  bl        U_MOD
  cmp       r0, #0
  bne       L6
  mov       r12, #1
L6:
  mov       r0, r12, LSL #0
  add       r13, r13, #8
  ldmia     r13!, {r15}
.end

.routine U_MOD
  stmdb     r13!, {r14, r2}
  movs      r2, r1, LSL #0
  beq       L7
  mov       r14, #0
  cmp       r2, r0, LSR #0x10
  movls     r2, r2, LSL #0x10
  cmp       r2, r0, LSR #8
  movls     r2, r2, LSL #8
  cmp       r2, r0, LSR #1
  bhi       L8
  cmp       r2, r0, LSR #2
  bhi       L9
  cmp       r2, r0, LSR #3
  bhi       L10
  cmp       r2, r0, LSR #4
  bhi       L11
  cmp       r2, r0, LSR #5
```

Listing A.4: Assemblercode des Testprogramms `prime`

```
    bhi     L12
    cmp     r2, r0, LSR #6
    bhi     L13
    cmp     r2, r0, LSR #7
    bhi     L14
L15:
    cmp     r0, r2, LSL #7
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #7
    cmp     r0, r2, LSL #6
L14:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #6
    cmp     r0, r2, LSL #5
L13:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #5
    cmp     r0, r2, LSL #4
L12:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #4
    cmp     r0, r2, LSL #3
L11:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #3
    cmp     r0, r2, LSL #2
L10:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #2
    cmp     r0, r2, LSL #1
L9:
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #1
L8:
    cmp     r0, r2, LSL #0
    adc     r14, r14, r14, LSL #0
    subcs   r0, r0, r2, LSL #0
    cmp     r1, r2, LSL #0
    movcc   r2, r2, LSR #8
    bcc     L15
    mov     r1, r14, LSL #0
    ldmia   r13!, {r15, r2}
L7:
    mov     r0, #0
    ldmia   r13!, {r15, r2}
.end
```

```
.routine _even
  stmdb     r13!, {r14}
  sub       r13, r13, #4
  str       r0, [r13]
  mov       r0, #2
  ldr       r1, [r13]
  bl        _divides
  add       r13, r13, #4
  ldmia     r13!, {r15}
.end
```

Listing A.5: Assemblercode des Testprogramms `prime` (Fortsetzung)

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[Abs] AbsInt Angewandte Informatik GmbH: *PAG in a Nutshell - Manual for the unexperienced User.*

[Abs02] AbsInt Angewandte Informatik GmbH: *The Program Analyzer User´s Manual*, 2002.

[aiS] http://www.aiSee.com.

[All70] Allen, F. E.: *Control flow analysis.* SIGPLAN Notices, pages 1–19, 1970.

[AMW95] Alt, Martin, Florian Martin, and Reinhard Wilhelm: *Generating Analyzers with PAG.* Technischer Bericht A 10/95, Universität des Saarlandes, 1995.

[ARM99] ARM Limited: *ARM Instruction Set Quick Reference Card*, 1999.

[ARM00] ARM Limited: *ARM Architecture Reference Manual*, 2000.

[ARM01] ARM Limited: *ARM Developer Suite Assembler Guide*, 2001.

[BC85] Bergeretti, Jean-Francois and Bernard Carré: *Information-Flow and Data-Flow Analysis of While-Programs.* ACM Transactions on Programming Languages and Systems, 7(1):37–61, 1985.

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

[CC92] Cousot, Patrick and Radhia Cousot: *Abstract interpretation frameworks.* Journal of Logic and Computation, 2(4):511–547, 1992.

[FKL+99] FERDINAND, CHRISTIAN, DANIEL KÄSTNER, MARC LANGENBACH, FLORIAN MARTIN, MICHAEL SCHMIDT, JÖRN SCHNEIDER, HENRIK THEILING, STEPHAN THESING, and REINHARD WILHELM: *Run-Time Guarantees for Real-Time Systems — The USES Approach.* In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, 1999.

[KL88] KOREL, BOGDAN and JANUSZ LASKI: *Dynamic program slicing.* Information Processing Letters, 29(3):155–163, 1988.

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

[KW02] KÄSTNER, DANIEL and STEPHAN WILHELM: *Generic control flow reconstruction from assembly code.* ACM SIGPLAN Notices, 37(7):46–55, 2002.

[Mar95] MARTIN, FLORIAN: *Entwurf und Implementierung eines Generators für Datenflußanalysatoren.* Diplomarbeit, Lehrstuhl für Programmiersprachen und Übersetzerbau, Universität des Saarlandes, 1995.

[Mar99] MARTIN, FLORIAN: *Generating Program Analyzers.* PhD thesis, Universität des Saarlandes, 1999.

[NNH99] NIELSON, FLEMMING, HANNE RIIS NIELSON, and CHRIS HANKIN: *Principles of Program Analysis.* Springer-Verlag, 1999.

[OO84] OTTENSTEIN, KARL J. and LINDA OTTENSTEIN: *The program dependence graph in a software development environment.* ACM SIGPLAN Notices, 19(5):177–184, 1984.

[RB89] REPS, THOMAS and THOMAS BRICKER: *Illustrating Interference in Interfering Versions of Programs.* Technical Report CS-TR-1989-827, University of Wisconsin, Madison, 1989.

[Sic97] SICKS, MARTIN: *Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches.* Diplomarbeit, Universität des Saarlandes, 1997.

[SP81] SHARIR, M. and A. PNUELI: *Two approaches to interprocedural data flow analysis.* Prentice-Hall, 1981.

[Tip95] TIP, FRANK: *A survey of program slicing techniques.* Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1995.

[TMSS03] THEILING, HENRIK, FLORIAN MARTIN, JÖRN SCHNEIDER, and MICHAEL SCHMIDT: *Specification of the Standard for a File Format used for Exchanging Results of Different Parts of a Run-Time Analysis (ERD).* Technischer Bericht, Universität des Saarlandes, AbsInt Angewandte Informatik GmbH, 2003.

[Wei79] Weiser, Mark: *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[Wei82] Weiser, Mark: *Programmers Use Slicing When Debugging*. Communications of the ACM, 25(7):446–452, 1982.

[Wei84] Weiser, Mark: *Program Slicing*. IEEE Transactions on Software Engineering, SE-10(4):352–357, 1984.

[Wil01] Wilhelm, Stephan: *Generische Rekonstruktion von Kontrollflußgraphen aus Assemblerprogrammen*. Diplomarbeit, Universität des Saarlandes, 2001.

[WM96] Wilhelm, Reinhard und Dieter Maurer: *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Verlag, 2. Auflage, 1996.