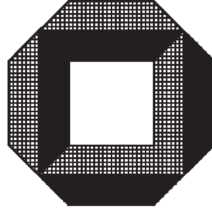


Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Dr. G. Goos

Studienarbeit

# Explizite Interprozedurale Abhängigkeitsgraphen

Hubert Schmid

*Betreuender Mitarbeiter*
Götz Lindenmaier

*Verantwortlicher Betreuer*
Prof. Dr. Gerhard Goos

Juni 2002

# Inhaltsverzeichnis

**1 Einleitung** **5**

1.1 Interprozedurale Analysen . . . . . . . . . . . . . . . . . . . . . 5

1.2 Firm und Libfirm . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3 Aufgabenstellung . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Interprozedurale Abhängigkeiten** **11**

2.1 Aufrufrelation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Abhängigkeiten der Methodenaufrufe . . . . . . . . . . . . . . . 14

2.3 Weitere Abhängigkeiten . . . . . . . . . . . . . . . . . . . . . . 17

2.4 Sonderfälle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 Darstellung** **21**

3.1 Realisierung der Sichten . . . . . . . . . . . . . . . . . . . . . . 21

3.2 Aufbau der Darstellung . . . . . . . . . . . . . . . . . . . . . . . 31

3.3 Abbau der Darstellung . . . . . . . . . . . . . . . . . . . . . . . 34

**4 Analyse der Aufrufrelation** **35**

4.1 Freie Methoden . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.2 Analyse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**5 Zusammenfassung** **39**

**Firm-Operationen** **41**

**Literaturverzeichnis** **43**

# Kapitel 1

# Einleitung

Abstraktion, Kapselung, Vererbung, Polymorphismus und Wiederverwendbarkeit sind Schlagwörter moderner Softwareentwicklungsmethoden. Diese Techniken versprechen, die Entwicklungszeit, sowie die Entwicklungs- und Wartungskosten für Programme zu reduzieren. Insbesondere werden diese Techniken in der objekt-orientierten Programmierung angewendet.

Auf der anderen Seite behindern diese Techniken aber häufig die Leistung von optimierenden Übersetzern. So arbeiten Programme meist wesentlich effizienter, wenn sie mit klassischen Methoden erstellt werden. Die Gründe hierfür sind vielseitig. So führt beispielsweise die Kapselung von Daten zu vielen kleinen Zugriffsprozeduren und entsprechend zu wesentlich mehr Prozeduraufrufen als in der klassischen Entwicklung. Abstraktion und Wiederverwendbarkeit führen dazu, dass Methoden häufig allgemeiner formuliert sind, als das für das eigentliche Programm notwendig ist.

Übersetzer, die sich im wesentlichen auf prozedurglobale Optimierungen beschränken, erzielen daher für die kürzeren und abstrakteren Prozeduren schlechtere Ergebnisse als in herkömmlichen Programmen. Durch den offenen Einbau von Prozeduren versuchen sie einige der Prozeduraufrufe einzusparen und die Analyseergebnisse zu verbessern. In objekt-orientierten Programmen wird der offene Einbau von Prozeduren jedoch häufig durch polymorphe Methodenaufrufe eingeschränkt.

## 1.1 Interprozedurale Analysen

Programmanalysen nennt man *intraprozedural*, wenn sie sich auf die Untersuchung einzelner Prozeduren beschränken und Prozeduraufrufe als atoma-

re Operationen mit unbekanntem Verhalten behandeln. Dagegen bezeichnet man Analysen als *interprozedural*, wenn sie über die Grenzen von Prozeduraufrufen hinweg arbeiten. Berücksichtigen sie zusätzlich einen beschränkten Ausführungskontext, so nennt man sie auch *kontextsensitive interprozedurale Analysen*.

Man verspricht sich, Ineffizienzen in der Programmausführung, die auf die modernen Softwareentwicklungsmethoden zurückzuführen sind, durch kontextsensitive interprozedurale Analysen beseitigen zu können. Beispielsweise hat *Trapp* in seiner Dissertation [3] nachgewiesen, dass durch diese Techniken wesentlich bessere Ergebnisse für objekt-orientierte Programme erzielt werden können.

Kontextsensitive Analysen untersuchen die Operationen des Programms unter verschiedenen Ausführungskontexten und erzielen daher in der Regel genauere Ergebnisse. Dafür sind sie aber einerseits in der Ausführung wesentlich langsamer als insensitive Analysen, und andererseits benötigen sie auch wesentlich mehr Speicher für die kontextabhängigen Analyseergebnisse. Für moderne optimierende Übersetzer ist daher eine geeignete Darstellung der interprozeduralen Abhängigkeiten von zentraler Bedeutung.

## 1.2 Firm und Libfirm

Diese Arbeit basiert auf der Zwischensprache Firm[1], die zuerst von Armbruster und von Roques [4] implementiert wurde. Es wird angenommen, dass der Leser mit ihr und ihrer Implementierung in der Bibliothek Libfirm vertraut ist. Zur Vollständigkeit wird aber eine kurze Einführung gegeben. Für eine ausführliche Beschreibung von Firm wird auf [1] verwiesen. Die Bibliothek Libfirm ist in [2] beschrieben.

Firm ist eine Zwischensprache zur übersetzerinternen Repräsentation von Programmen. Sie wurde mit dem Ziel entwickelt, effiziente Optimierungen von objekt-orientierten Programmen zu ermöglichen, und basiert auf der statischen Einmalzuweisung[2] (SSA). Neben den üblichen Datenstrukturen für Typen und Definitionen wird jede Methode als Prozedurgraph dargestellt, in dem die Operationen die Knoten und die Steuer- und Datenflussabhängigkeiten zwischen den Operationen die Kanten sind. Insbesondere werden lokale, alias-freie Variablen zu Datenflusskanten aufgelöst. Durch *Block*-Knoten werden Operationen zu Grundblöcken zusammengefasst. Die

[1] Abkürzung für „Firm Intermediate Representation Mesh“
[2] engl.: static single assignment

Operationen selbst orientieren sich an dem Befehlssatz einer fiktiven RISC-Architektur. Andererseits unterstützt Firm beispielsweise aber auch die explizite Darstellung von Ausnahmen, sowie Operationen für polymorphe Methodenaufrufe.

Jede Operation liefert in Firm per Definition genau ein Ergebnis. Mehrere Werte können allerdings in Tupeln zusammengefasst werden. Die *Proj*-Operationen – die in dieser Arbeit eine wesentliche Rolle spielen – extrahieren einen Wert aus einem Tupel.

Abbildung 1.1 zeigt einen Firm-Graphen für eine Methode, die ein *int*-Argument nimmt und dessen Quadrat zurückgibt. Die Grundblöcke sind implizit durch die gerasterten Rechtecke dargestellt. Die Pfeile zwischen Operationen sind Datenflussabhängigkeiten von den Verwendungen auf ihre Definitionen. Steuerflussabhängigkeiten sind durch die Pfeile von Grundblöcken auf die Operationen dargestellt, die den Steuerfluss ändern.

Jeder Graph besitzt jeweils genau eine *Start*- und eine *End*-Operation mit zugehörigen Grundblöcken, die im wesentlichen nur diese Operationen enthalten. Die beiden Blöcke werden entsprechend als *Start*- und *End*-Block bezeichnet. Die *Start*-Operation gibt den ersten auszuführenden Grundblock

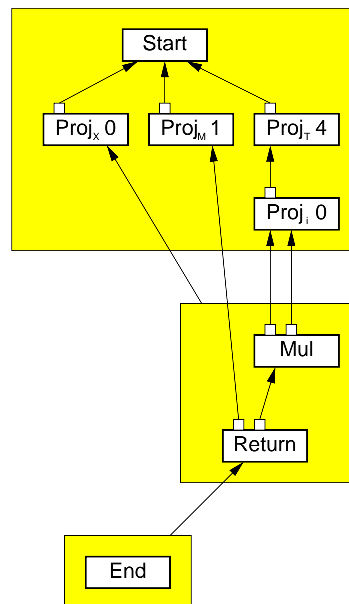



Abbildung 1.1: Prozedurgraph in Firm für eine Methode, die das Quadrat ihres Arguments zurückgibt.

der Methode an und liefert als Ergebnis ein Tupel mit dem Speicherzustand und den Werten der Parameter beim Aufruf. Die *Return*-Operation beendet die Ausführung der Methode und kehrt mit dem Speicherzustand und dem Ergebnis der Multiplikation zum Aufrufer zurück. Der *End*-Block hat alle Operationen als Vorgänger, die die Methoden beenden. Dieser unechte Steuerzusammenfluss vereinfacht im wesentlichen die Darstellung und Implementierung der Zwischensprache.

Die Bibliothek LIBFIRM ist eine Implementierung der Zwischensprache FIRM in der Programmiersprache C [5]. Die Implementierung basiert dabei auf objekt-orientierten Techniken. Die FIRM-Operationen werden als Objekte dargestellt, in denen u. a. die Steuer- und Datenflussvorgänger in einer Reihung mit Zeigern gespeichert werden. Insbesondere ist daher die Navigation in Prozedurgraphen im allgemeinen nur gegen den Steuer- bzw. Datenfluss möglich. Auf die Attribute von Operationen wird sowohl mit Zugriffsfunktionen als auch direkt zugegriffen.

## 1.3 Aufgabenstellung

Datenflussabhängigkeiten zwischen lokalen Variablen und intraprozedurale Steuerflussabhängigkeiten werden in FIRM explizit durch Kanten im Prozedurgraphen dargestellt. Interprozedurale Abhängigkeiten sind jedoch nur über den Speicher oder implizit über die Semantik von Operationen beschrieben. Die explizite Darstellung der intraprozeduralen Abhängigkeiten erlaubt es, effiziente intraprozedurale Analysen einfach zu formulieren. Interprozedurale Analysen müssen hingegen den Steuerfluss der Methodenaufrufe, sowie die Datenflussabhängigkeiten der formalen Parameter und der Ergebniswerte der aufgerufenen Methode besonders behandeln.

Das Ziel dieser Arbeit ist es daher, die interprozeduralen Abhängigkeiten von Methodenaufrufen explizit durch Kanten zwischen den Prozedurgraphen darzustellen, um die interprozeduralen Analysen zu vereinfachen. Insbesondere muss dabei die SSA-Form erhalten bleiben. Um die interprozeduralen Abhängigkeiten darstellen zu können, wird die Zwischensprache FIRM um zusätzliche Operationen und Semantik erweitert.

Die Darstellung wird im Rahmen dieser Arbeit als Erweiterung der Bibliothek LIBFIRM implementiert. Damit existierende Programmanalysen und -transformationen einfach auf diese Erweiterung angepasst werden können, müssen zusätzliche Anforderungen an die Implementierung erfüllt werden:

- Die Implementierung bietet sowohl eine intra- als auch eine interprozedurale Darstellung des Programms. Diese bezeichnen wir auch als *Sichten* auf die Programmrepräsentation. Die beiden Sichten unterscheiden sich vor allem in der Darstellung der interprozeduralen Abhängigkeiten.

- Die Sichten entsprechen von der Struktur und der Schnittstelle der bisherigen Darstellung in der Libfirm, damit existierende Analysen und Transformationen einfach auf diese Sichten übertragen werden können.

- Insbesondere können Änderungen und Transformationen eines Programms über beide Sichten durchgeführt werden. Diese müssen sich in der jeweils anderen Sicht konsistent wiederspiegeln.

Die Aufrufe von Methoden sind in objekt-orientierten Programmen häufig polymorph. An einer statischen Aufrufstelle können daher abhängig vom Ausführungskontext im allgemeinen verschiedene Methoden aufgerufen werden. Daher muss die Programmrepräsentation insbesondere für die Darstellung der interprozeduralen Steuer- und Datenflussabhängigkeiten von polymorphen Aufrufstellen geeignet sein. Die Arbeit orientiert sich dabei an der Beschreibung interprozeduraler Abhängigkeitsgraphen in der Dissertation von Trapp [3].

# Kapitel 2

# Interprozedurale Abhängigkeiten

In diesem Kapitel werden die Steuer- und Datenflussabhängigkeiten betrachtet, die zwischen Methoden existieren. Da interprozeduraler Steuerfluss in FIRM nur durch Methodenaufrufe dargestellt werden kann, wird zunächst auf die Beziehung zwischen Aufrufstellen und aufgerufenen Methoden eingegangen. Im darauf folgenden Abschnitt betrachten wir dann die interprozeduralen Abhängigkeiten dieser Methodeaufrufe. Schließlich gehen wir noch auf einige Sonderfälle ein, die im Aufbau besonders behandelt werden.

## 2.1 Aufrufrelation

Unter der *Aufrufrelation* für ein Programm verstehen wir die Beziehung zwischen statischen Programmstellen und Methoden, die ausdrückt, welche Methoden an welchen Programmstellen aufgerufen werden. Bezeichnen wir genauer die Aufrufrelation mit *calls*, die Menge der Methoden mit $M$ und die Menge der statischen Programmstellen, an denen Methodenaufrufe erfolgen können, mit $C$, so können wir die Aufrufrelation als Teilmenge des kartesischen Produktes $C \times M$ auffassen. Für eine Programmstelle $c \in C$ und eine Methode $m \in M$ gilt dann $(c, m) \in \mathit{calls}$ genau dann, wenn es mindestens eine Programmausführung gibt, so dass die Methode $m$ an der Programmstelle $c$ mindestens einmal aufgerufen wird.

Es ist theoretisch nicht entscheidbar, ob eine Methode an einer Programmstelle aufgerufen werden kann.[1] Insbesondere ist daher die Aufrufrelation

[1] Es ist bereits unentscheidbar, ob die Programmstelle überhaupt erreichbar ist.

unentscheidbar. Zur Darstellung der interprozeduralen Abhängigkeiten interessieren wir uns daher immer nur für eine pessimistische Abschätzung, d. h. für eine Obermenge der Aufrufrelation.

Für eine Methode $m \in M$ nennen wir die Programmstellen, an denen die Methode aufgerufen werden kann, die Aufrufer von $m$. Die Methoden, die von einer Programmstelle $c \in C$ aufgerufen werden können, nennen wir entsprechend die Aufgerufenen (Methoden) von $c$. Bezüglich einer Abschätzung *calls* der Aufrufrelation entsprechen die Aufrufer von $m$ der Menge $\{c \in C : (c, m) \in \mathit{calls}\}$ und die Aufgerufenen von $c$ der Menge $\{m \in M : (c, m) \in \mathit{calls}\}$.

## Methoden und Aufrufstellen

In Firm können Methodenaufrufe nur an *Call*-Operationen erfolgen. Die *Call*-Operationen, die in der Zwischenrepräsentation des Programms explizit dargestellt sind, nennen wir daher *interne Aufrufstellen* und bezeichnen die Menge dieser Operationen mit $C_i$. Entsprechend werden die Methoden, die in der Zwischensprache dargestellt sind, *interne Methoden* genannt und ihre Menge mit $M_i$ bezeichnet.

Neben den internen Methoden können auch Methoden in getrennten Übersetzungseinheiten oder in der Laufzeitumgebung des Systems existieren. Umgekehrt können auch Methoden von Programmstellen außerhalb der Zwischenrepräsentation aufgerufen werden. Daher werden die Methoden aus $M$, die nicht zu den internen Methoden gehören, als *externe Methoden* bezeichnet. Die Programmstellen außerhalb des dargestellten Programms, an denen Methoden aufgerufen werden können, werden entsprechen *externe Aufrufstellen* genannt. Zu den externen Aufrufstellen zählen wir insbesondere den Aufrufer des Hauptprogramms, das in Firm auch eine Methode ist.

Wir nehmen an, dass das Programmverhalten der externen Methoden und Aufrufstellen zur Übersetzungszeit nicht bekannt ist. Ihre Implementierung ist im allgemeinen nicht einmal statisch fest, sondern kann von Programmlauf zu Programmlauf variieren. Ihre Anzahl ist daher theoretisch unbeschränkt. Wir führen daher eine abstrakte Methode $m_u$ und eine abstrakte Aufrufstelle $c_u$ ein, deren Programmverhalten beliebig und unbekannt ist. Insbesondere können wir mit diesen die externen Methoden und Aufrufstellen abschätzen. Wir werden sie aber auch zur Abschätzung von internen Methoden und Aufrufstellen verwenden.

Schließlich betrachten wir noch die *Call*-Operationen genauer. Diese bestimmen die aufzurufende Methode durch den Wert des Funktionszeigers, den sie

als Argument erwarten. Da in FIRM Arithmetik auf Funktionszeigern möglich ist, können diese im allgemeinen beliebige Werte annehmen, – insbesondere auch *ungültige* Werte, die nicht der Adresse einer Methode entsprechen. In FIRM ist der Sprung in den Rumpf einer internen Methode mit einer *Call*-Operation undefiniert. Deshalb können wir die Ausführung einer *Call*-Operation mit einem ungültigen Wert durch einen Aufruf der abstrakten Methode $m_u$ abschätzen.

## Innere Abschätzung

Die interprozeduralen Abhängigkeiten zwischen externen Methoden können nicht explizit dargestellt werden. Wir reduzieren daher die Abschätzung der Aufrufrelation auf die Beziehungen von internen Methoden und internen Aufrufstellen. Die Unterscheidung der externen Methoden bzw. der externen Aufrufstellen ist nach den bisherigen Annahmen nicht sinnvoll. Stattdessen abstrahieren wir sie durch die unbekannte Methode $m_u$ und die unbekannte Aufrufstelle $c_u$.

Wir setzen $C' := C_i \cup \{c_u\}$ und $M' := M_i \cup \{m_u\}$. Eine *innere Abschätzung* $calls'$ der Aufrufrelation für ein Programm ist eine Relation über $C' \times M'$, für die folgende Bedingungen erfüllt sind:

1. Ist $calls : C \times M$ die Aufrufrelation, $c \in C$ eine Aufrufstelle und $m \in M$ eine Methode mit $(c, m) \in calls$, so gilt:
   (a) $c \in C_i \implies (c, m) \in calls'$ oder $(c, m_u) \in calls'$
   (b) $m \in M_i \implies (c, m) \in calls'$ oder $(c_u, m) \in calls'$
2. Ist $c \in C_i$ eine *Call*-Operation und gibt es eine Programmausführung, so dass die Operation mindestens einmal mit einem ungültigen Funktionszeiger $p$ ausgeführt wird, d. h. $p \notin M$, so gilt: $(c, m_u) \in calls'$.

Die Anzahl der internen Methoden sowie die Anzahl der internen Aufrufstellen sind im allgemeinen linear in der Programmlänge. Man kann auch eine Familie von Programmen konstruieren, so dass die Größe der Aufrufrelation quadratisch zur Programmlänge wächst. Wir nehmen allerdings an, dass dies praktisch nicht der Fall ist, und dass sich die Größe der Aufrufrelation in der Regel linear zur Programmlänge verhält.

Die Abschätzung kann andererseits aber auch verkleinert werden, indem weitere interne Methoden und Aufrufstellen durch $m_u$ bzw. $c_u$ abgeschätzt werden. So ist insbesondere die Relation, die durch die Menge

$$\{(c, m_u) : c \in C_i\} \cup \{(c_u, m) : m \in M_i\}$$

gegeben ist, eine korrekte innere Abschätzung, deren Größe immer linear in der Programmlänge ist.

## 2.2 Abhängigkeiten der Methodenaufrufe

In diesem Abschnitt werden die Steuer- und Datenflussabhängigkeiten von Methodenaufrufen erklärt. In der interprozeduralen Darstellung beschränken wir uns auf diese Abhängigkeiten. Auf weitere interprozedurale Abhängigkeiten gehen wir im nächsten Abschnitt kurz ein.

Zunächst beschreiben wir aber noch die Aufteilung der Grundblöcke beim Methodenaufruf, die wir in Abschnitt 3.1 weiter verschärfen werden. Die *Call*-Operation für den Methodenaufruf wird in FIRM intraprozedural als atomare Operation behandelt. Insbesondere ändert sie den Steuerfluss nicht, – abgesehen davon, dass der Methodenaufruf eine Ausnahme werfen kann. Interprozedural wird aber zwischen den Operationen vor und nach[2] einer Aufrufstelle der Rumpf der aufgerufenen Methode ausgeführt.

In einer interprozeduralen Darstellung können daher eine Operation für den Methodenaufruf und Operationen, die von dieser abhängen, nicht im selben Grundblock stehen. Um dies zu vermeiden muss gegenfalls nach der Aufrufstelle ein neuer Grundblock eingefügt und die abhängigen Operationen in diesen verschoben werden. Wir setzen diese Aufteilung in der Beschreibung der interprozeduralen Abhängigkeiten im folgenden stets voraus.

Für eine Methode und eine Aufrufstelle aus der inneren Abschätzung unterscheiden wir die folgenden drei Arten interprozeduraler Abhängigkeiten, die explizit dargestellt werden:

1. Abhängigkeiten für den Aufruf einer Methode:
   (a) Der Steuerfluss von der Aufrufstelle auf den ersten auszuführenden Grundblock der aufgerufenen Methode.
   (b) Die Datenflussabhängigkeiten der formalen Parameter der aufgerufenen Methode auf die Argumente der *Call*-Operation, sowie die Datenflussabhängigkeit für den Speicherzustand.
2. Abhängigkeiten für den regulären Rücksprung aus einer Methode:

[2] „vor“ und „nach“ ist bezüglich der partiellen Ordnung zu verstehen, die durch die Datenflussabhängigkeiten im Grundblock induziert wird.

   (a) Der Steuerfluss von einer *Return*-Operation auf den Grundblock, der intraprozedural dem Methodenaufruf folgt.
   (b) Die Datenflussabhängigkeiten von den Ergebnissen des Methodenaufrufes auf die Rückgabewerte der *Return*-Operation, sowie die Datenflussabhängigkeit für den Speicherzustand.

3. Abhängigkeiten beim Abbruch einer Methode durch das Werfen einer Ausnahme, die im Rumpf der Methode nicht gefangen wird:
   (a) Der Steuerfluss von der Operation, die die Ausnahme wirft, auf den Grundblock beim Aufrufer, in dem die Ausnahme behandelt wird.
   (b) Die Datenflussabhängigkeit von der Verwendung des Speicherzustandes in der Ausnahmebehandlung auf die Definition des Speicherzustandes beim Werfen der Ausnahme.

Abbildung 2.1 zeigt diese Abhängigkeiten. Auf der linken Seite ist der Grundblock eines Methodenaufrufes zusammen mit dem Grundblock für die Ausnahmebehandlung dargestellt. Die rechte Seite zeigt den Anfang und das Ende der aufgerufenen Methode. Die intraprozeduralen Abhängigkeiten des Methodeaufrufes sind mit durchgezogenen Pfeilen dargestellt – die interprozeduralen Abhängigkeiten durch gestrichelte Pfeile. Die Aufteilung des Grundblocks ist auf der linken Seite durch die gestrichelten Linien angedeutet.

Im allgemeinen können sowohl an einer Aufrufstelle mehrere Methoden, als auch eine Methode von mehreren Aufrufstellen aufgerufen werden. Die interprozeduralen Datenflussabhängigkeiten über die unterschiedlichen Steuerflussvorgänger müssen dann in *Phi*-Operationen in dem entsprechenden Grundblock zusammengeführt werden, um die SSA-Form zu erhalten.

Der Aufwand für diese Darstellung kann unnötig komplex werden. Nehmen wir an, dass eine Methode von $m$ verschiedenen Aufrufstellen aufgerufen werden kann, und dass die Methode $n$ Operationen enthält, die die Methode beenden, d. h. *Return*-Operationen bzw. Operationen, die eine im Rumpf der Methode nicht behandelte Ausnahme werfen. Dann werden für diese Methode $m \cdot n$ Steuerflussabhängigkeiten und entsprechend viele Datenflussabhängigkeiten dargestellt. Im allgemeinen können $m$ und $n$ sehr groß sein, – insbesondere da Elementzugriffe und Methodenaufrufe in der Regel auch Ausnahmen werfen können.

Um den Aufwand zu reduzieren und die Darstellung zu vereinfachen, fügen wir sowohl für das reguläre Beenden einer Methode als auch für den Abbruch

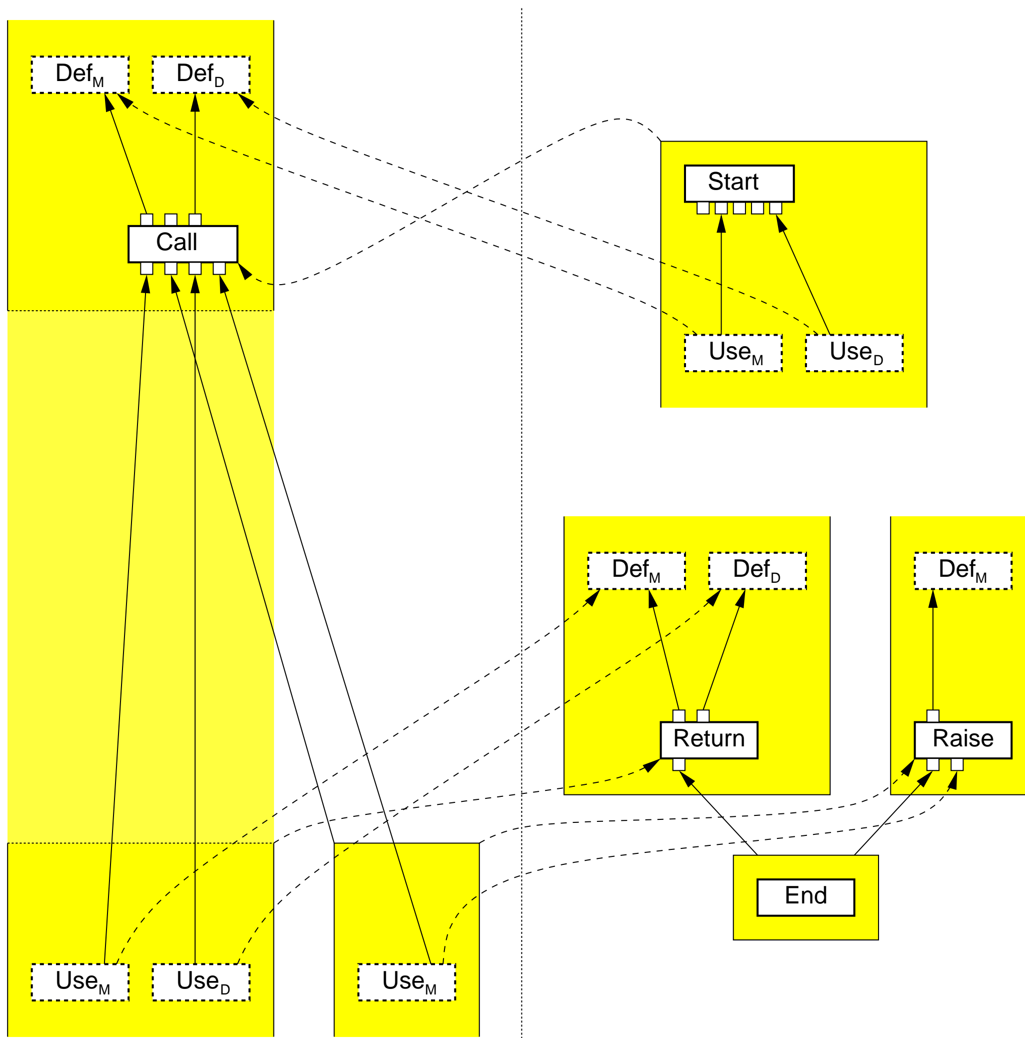



Abbildung 2.1: Abhängigkeiten eines Methodenaufrufes

einer Methode durch eine geworfene Ausnahme je einen neuen Grundblock – d. h. einen künstlichen Steuerzusammenfluss – in die Methode ein. Im ersten Grundblock werden alle *Return*-Operationen der Methode zusammengefasst, und die Rückgabewerte in *Phi*-Operationen vereinigt. Beim Abbruch durch eine Ausnahme gibt es keine Rückgabewerte. Im zweiten Grundblock, der die ausnahmewerfenden Operationen als Steuerflussvorgänger besitzt, werden daher nur die Speicherzustände der entsprechenden Operationen in einer *Phi*-Operation zusammengefasst. Damit reduziert sich der Aufwand für die Darstellung der interprozeduralen Steuerflussvorgänger für eine Methode von

$m \cdot n$ auf $2\,m + n$. Das Entsprechende gilt auch für die Anzahl der interprozeduralen Datenflussabhängigkeiten.

In Firm existiert bereits ein künstlicher Steuerzusammenfluss am Ende jeder Methode, – der sogenannte *End*-Block. Wir können diesen Block aber nicht für unsere Zwecke verwenden, da er sowohl die regulären Rücksprungoperationen, als auch die Operationen, die Ausnahmen werfen können, als Steuerflussvorgänger besitzt, aber nur im ersten Fall Rückgabewerte existieren.

## 2.3 Weitere Abhängigkeiten

Im letzten Abschnitt wurden die interprozeduralen Steuer- und Datenflussabhängigkeiten von Methodenaufrufen erklärt. Da interprozeduraler Steuerfluss in Firm nur durch *Call*-Operationen entsteht, ist dieser bereits vollständig beschrieben. Neben den Datenflussabhängigkeiten von Methodenaufrufen gibt es im allgemeinen aber noch weitere interprozedurale Abhängigkeiten.

**Programm 2.1** Beispiel in C

```
int a;

void f(void) {
    a := 1;
}

int main(void) {
    f();
    return a;
}
```

Programm 2.1 zeigt ein Beispiel in der Quellsprache C [5]. Die Methode `f` definiert den Wert von `a`, den das Hauptprogramm anschließend verwendet. Diese interprozedurale Datenflussabhängigkeit steht aber nicht in direktem Zusammenhang mit dem Methodenaufruf von `f`.

Neben den interprozeduralen Datenflussabhängigkeiten über globale Variablen sind insbesondere die Abhängigkeiten über Haldenobjekte von Bedeutung. Da die Behandlung dieser Abhängigkeiten im allgemeinen sehr komplex

ist, beschränkt sich diese Arbeit auf die Darstellung der interprozeduralen Datenflussabhängigkeiten, die im letzten Abschnitt erklärt wurden. Die anderen Datenflussabhängigkeiten sind dabei nur implizit über den abstrakten Speicherzustand modelliert.

## 2.4 Sonderfälle

In diesem Abschnitt werden vier Sonderfälle von Aufrufbeziehungen betrachtet, die in der Realisierung der Darstellung besonders behandelt werden. In den ersten beiden Fällen ist eine interprozedurale Darstellung wie im allgemeinen Fall nicht möglich. In den anderen beiden Fällen bringt die explizite Darstellung der Abhängigkeiten keine Vorteile, sondern erhöht nur deren Aufwand.

Es kann Methoden geben, die keine Aufrufer haben. Zum Beispiel können aufgrund der Wiederverwendbarkeit für Klassen Zugriffsfunktionen implementiert sein, die im eigentlichen Programm nie aufgerufen werden. Aber auch vorausgehende Optimierungen können dazu führen, dass Methoden nicht mehr aufgerufen werden. Wird zum Beispiel eine Methode an sämtlichen Aufrufstellen offen eingebaut, so besitzt die Methode selbst keinen Aufrufer mehr. Für diese Methoden muss kein Code erzeugt werden und sie können aus der Zwischenrepräsentation entfernt werden.

**Programm 2.2** Beispiel in Java

```
interface I {
    void f();
}

class C {
    void g(I i) {
        i.f();
    }
}
```

Andererseits muss auch der Fall betrachtet werden, dass an einer Aufrufstelle keine Methode aufgerufen werden kann. Programm 2.2 zeigt ein Ausschnitt eines Quellprogramms in der Programmiersprache Java [6]. Gibt es im vollständigen Programm keine Klasse, die die Schnittstelle I und die poly-

morphe Methode `f` implementiert, so kann an der Aufrufstelle in der Methode `g` auch keine Methode aufgerufen werden.

Die *Call*-Operation kann in diesem Fall nicht ausgeführt werden und ist daher auch nicht erreichbar. Sie kann also in FIRM durch eine *Bad*-Operation ersetzt werden. Im obigen Beispiel wirft spätestens der Zugriff auf die Entität `f` von `i` eine Ausnahme.

Wir betrachten schließlich noch die beiden Fälle, in denen es nur unbekannte Aufrufer, bzw. aufgerufene Methoden gibt. Zum Beispiel besitzt das Hauptprogramm nur den externen – also unbekannten – Aufrufer, wenn es nicht rekursiv ist. Andererseits können an Aufrufstellen auch Methoden aufgerufenen werden, die nur deklariert, aber nicht implementiert sind. In diesen Fällen bringt die explizite Darstellung der unbekannten Vorgänger keine Vorteile für die Datenflussanalyse gegenüber der intraprozeduralen Darstellung mit den implizit unbekannten Ergebnissen der *Call*- bzw. *Start*-Operationen.

# Kapitel 3

# Darstellung

In diesem Kapitel beschreiben wir die erweiterte Darstellung sowie deren Implementierung. Während wir im ersten Abschnitt auf die Realisierung der Sichten eingehen, skizzieren wir in den beiden anschließenden Abschnitten den Aufbau und Abbau der Sichten aus der bzw. in die intraprozedurale Darstellung.

## 3.1 Realisierung der Sichten

In Abschnitt 1.3 auf Seite 8 haben wir bereits beschrieben, welche Anforderungen an die intra- und und an die interprozedurale Sicht auf die Programmrepräsentation bestehen. Wir untersuchen die Anforderungen in diesem Abschnitt genauer und leiten daraus die wesentlichen Merkmale einer zu implementierenden Realisierung her.

Existierende, intraprozedurale Programmanalysen und -transformationen sollen einfach auf die neu eingeführte intraprozedurale Sicht angepasst werden können. Daher nehmen wir an, dass die bisherige Programmdarstellung der Libfirm im wesentlichen mit der intraprozeduralen Sicht übereinstimmt. Andererseits sollen diese Verfahren aber auch einfach auf die interprozedurale Sicht erweitert werden können. Wir nehmen daher weiterhin an, dass die interprozedurale Sicht zumindest strukturell der intraprozeduralen Sicht sehr ähnlich ist.

Die Operationen der Zwischensprache werden in der Libfirm-Bibliothek als Objekte – bzw. Datenstrukturen – dargestellt, in denen Referenzen auf ihre Steuer- und Datenflussvorgänger gespeichert sind. Steuer- und Datenflussnachfolger werden in der Regel nicht explizit gespeichert. Datenflussanalysen

und Transformationen durchlaufen den Prozedurgraphen, in dem sie bei einer Menge von Operation starten und mit Tiefen- oder Breitensuche die Vorgänger besuchen.

Programmtransformationen können neue Operationen in den Prozedurgraphen einfügen, die Eigenschaften und insbesondere die Vorgänger von existierenden Operationen ändern, sowie Operationen implizit aus dem Prozedurgraphen löschen, indem alle Abhängigkeiten auf diese Operationen entfernt werden. Im allgemeinen basieren diese Änderungen aber immer auf einzelnen Operationen der Zwischensprache. Damit Änderungen konsistent in der jeweils anderen Sicht durchgeführt werden können, müssen sich die betroffenen Operationen in den beiden Sichten im wesentlichen entsprechen. Da wir die möglichen Änderungen auf den Sichtendarstellungen nicht unnötig stark einschränken wollen, können wir also annehmen, dass die meisten Operationen in den beiden Sichten sich entsprechen.

Für die Eigenschaften der Operationen existieren in LIBFIRM Zugriffsfunktionen. Indem diese Funktionen erweitert werden, können Änderungen an einer Sicht explizit in der jeweils anderen Sicht transparent durchgeführt werden. Allerdings greifen einige existierende Programmtransformationen teilweise direkt auf die Attribute der Objekte zu. Solche Änderungen können in der Sichtendarstellung jedoch nicht explizit in der jeweils anderen Sicht durchgeführt werden. Die Änderung dieser direkten Zugriffe in die entsprechenden Funktionsaufrufe ist aufwendig, und verringert die Effizienz der Verfahren.

Wir interessieren uns daher für eine Darstellung, die den direkten Zugriff und insbesondere die direkte Änderung der Attribute für die meisten Operationen bzw. Operationsarten weiterhin ermöglicht. Dafür müssen diese Operationen in den beiden Sichten sich nicht nur entsprechen, sondern sogar identisch sein. Insbesondere haben sie in den beiden Sichten die selben Steuer- und Datenflussvorgänger und gehören weiterhin den selben Grundblöcken an.

Wir müssen einige neue Operationsarten einfügen, um die Semantik von interprozeduralen Programmgraphen beschreiben zu können. Die Unterschiede der beiden Sichten realisieren wir, indem wir in einzelnen Operationen getrennte Attribute für den intra- und interprozeduralen Graphen speichern. Dabei werden wir uns vor allem auf die Vorgängerrelation beschränken. Die Zugriffsfunktionen für diese Operationen werden so implementiert, dass sie jeweils auf den Attributen der aktuellen Sicht arbeiten. Die ausgewählte Sicht bestimmen diese Funktionen über eine globale, für die gesamte Programmrepräsentation gültige Zustandsvariable. Da die direkte Änderung der Attribute dadurch nicht mehr erlaubt ist, beschränken wir uns dabei auf eine geringe Anzahl von Operationen bzw. Operationsarten.

### Steuerflussabhängigkeiten

Da es in Firm keine Operationen gibt, die explizit interprozeduralen Steuerfluss beschreiben, führen wir hierfür drei neue Operationsarten ein. Diese sind nur in der interprozeduralen Sicht erreichbar. Die *CallBegin*-Operation ersetzt die intraprozedurale *Call*-Operation. Sie besitzt als einziges Argument einen Funktionszeiger auf die aufzurufende Methode. Ihre Steuerflussnachfolger sind die *Start*-Blöcke der Methoden, die an dieser Aufrufstelle aufgerufen werden können.

Für den erfolgreichen Rücksprung aus einer Methode führen wir die *EndReg*-Operation ein. Sie nimmt keine Argumente und hat die Grundblöcke als Steuerflussnachfolger, die intraprozedural den Aufrufstellen der Methode folgen. Entsprechend führen wir die *EndExcept*-Operation ein, die den Abbruch einer Methoden durch das Werfen einer Ausnahme darstellt. Sie besitzt die Grundblöcke für die Ausnahmebehandlung an den Aufrufstellen als Steuerflussnachfolger.

Im Gegensatz zu den intraprozeduralen Operationen für den Aufruf einer Methode und für den Rücksprung aus der Methode – *Call* und *Return* – haben die interprozeduralen Steuerfluss-Operationen keine Übergabeparameter, da die entsprechenden Datenflussabhängigkeiten in der interprozeduralen Sicht explizit dargestellt sind.

Auf der anderen Seite betrachten wir die Operationen, die diese interprozeduralen Steuerfluss-Operationen als Vorgänger besitzen. In Firm haben aber nur Grundblöcke Steuerflussvorgänger. Da sich der Steuerfluss in den beiden Sichten unterscheidet, muss es also auch Grundblock-Knoten geben, die in der intra- und in der interprozedurale Sicht unterschiedliche Steuerflussvorgänger besitzen. Wir führen daher in gewisser Weise eine neue Grundblock-Operationsart ein, die Steuerflussvorgänger für beide Sichten speichert. Die zugehörigen Zugriffsfunktionen werden so angepasst, dass sie je nach ausgewählter Sicht auf den entsprechenden Vorgänger operieren.

Grundblock-Operationen nehmen in der Zwischensprache allerdings eine besondere Rolle ein. Sie sind bereits implizit durch ihre besondere Beziehung zu anderen Operationen als solche erkennbar, – und daher wird ihr Typ in der Regel nicht explizit geprüft. Wir führen daher keine neue Operationsart ein, sondern erweitern den existierenden Typ, so dass er entweder für die Sichten die selben Vorgänger besitzt, oder die Vorgänger für die beiden Sichten getrennt speichert.

Für die explizite Darstellung der interprozeduralen Abhängigkeiten können wir drei Arten von Grundblöcken unterscheiden, die in den beiden Sichten

unterschiedliche Steuerflussvorgänger besitzen:

1. Der erste Grundblock einer Methode hat in der intraprozeduralen Sicht keine Steuerflussvorgänger.[1] In der interprozeduralen Sicht besitzt er alle *CallBegin*-Knoten als Vorgänger, von denen die Methode aufgerufen werden kann.

2. Der Grundblock, der intraprozedural einem explizit dargestellten Methodenaufruf folgt, hat in der interprozeduralen Darstellung die *EndReg*-Operationen der aufgerufenen Methoden als Steuerflussvorgänger.

3. Entsprechend besitzt der Grundblock, der intraprozedural die Ausnahmen von einem Methodenaufruf behandelt, in der interprozeduralen Darstellung die *EndExcept*-Operationen der aufgerufenen Methoden als Steuerflussvorgänger.

Diese Darstellung der interprozeduralen Steuerflussabhängigkeiten schränkt einige Programmtransformationen ein. Beim Aufteilen von Grundblöcken müssen die Steuerflussvorgänger beider Sichten berücksichtigt werden. Grundblöcke mit unterschiedlichen Steuerflussvorgängern können nicht mit ihren Vorgängerblöcken vereinigt werden. Weiterhin können Operationen nur eingeschränkt über Grundblockgrenzen hinweg verschoben werden.

Um die Transformationen zu vereinfachen, die Grundblöcke vereinigen, führen wir eine neue Operation ein. Die *Jmp'*-Operation entspricht einer *Jmp*-Operation, die nicht optimiert werden darf, d. h. ihr Grundblock darf nicht mit dem nachfolgenden Grundblock verschmolzen werden. Wir werden diese Operation insbesondere dann einsetzen, wenn einer der beiden zugehörigen Grundblöcke interprozedural andere Steuerflussvorgänger besitzt.

## Datenflussabhängigkeiten

Die unterschiedlichen Steuerflussabhängigkeiten der beiden Sichten können damit vollständig dargestellt werden. In diesem Abschnitt befassen wir uns mit der Darstellung der interprozeduralen Datenflussabhängigkeiten, die wir in 2.2 auf Seite 14 aufgeführt haben. Wir betrachten dazu Operationen, die interprozedurale Datenflussvorgänger besitzen. Diese Operationen können mit wenigen Ausnahmen von jeder Operationsart sein. Im letzten Abschnitt

[1] Aus implementierungstechnischen Gründen besitzt der *Start*-Block einer Methode in Libfirm die *Start*-Operation der Methode als abstrakten Steuerflussvorgänger.

wurde aber bereits festgestellt, dass nur wenige Operationsarten für die Sichtendarstellung geändert werden sollen. Andererseits gilt für diese Operationen folgende Aussage:

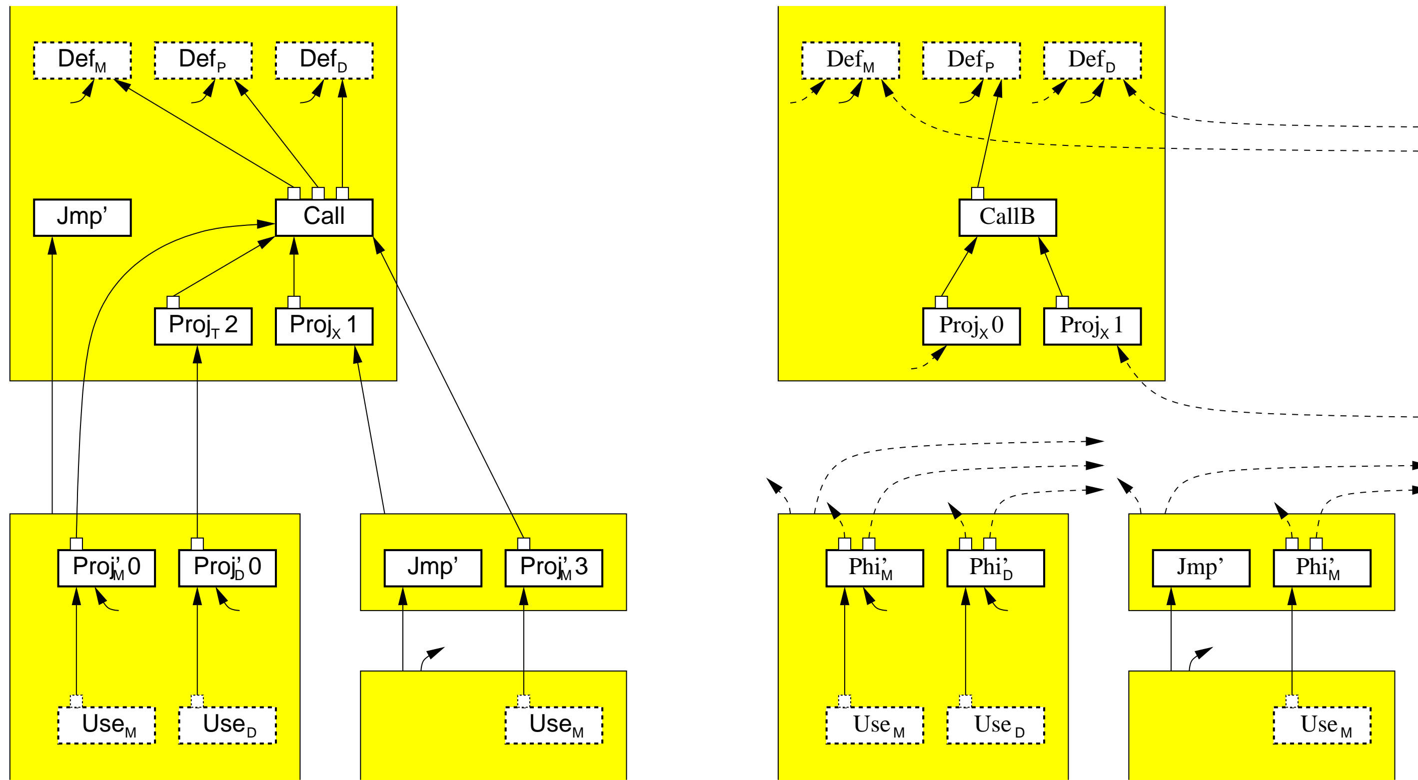
> Besitzt eine Operation einen interprozeduralen Datenflussvorgänger im Sinne von 2.2, so hängt sie in der intraprozeduralen Sicht von dem Ergebnis einer *Call*-Operation oder einer *Start*-Operation ab.

Die *Call*- und *Start*-Operationen haben immer mehrere Ergebnisse und fassen diese jeweils zu einem Tupel zusammen. Die oben beschriebenen Operationen hängen daher nie direkt von diesen ab, sondern immer über mindestens eine *Proj*-Operation, die den entsprechenden Wert aus dem Tupel extrahiert. Die *Proj*-Operationen selbst haben nur wenig Semantik. Es bietet sich daher an, diese Operationen durch neue Operationen zu ersetzen, die in der intraprozeduralen Sicht semantisch den *Proj*-Operationen entsprechen. In der interprozeduralen Sicht verweisen sie hingegen explizit auf die interprozeduralen Datenflussvorgänger.

Wir berücksichtigen weiterhin, dass an einer Aufrufstelle im allgemeinen mehrere Methoden aufgerufen werden können, bzw. dass eine Methode von mehreren Aufrufstellen aufgerufen werden kann. Die Datenflussabhängigkeiten über die verschiedenen Steuerflussvorgänger müssen dann in *Phi*-Operationen zusammengefasst werden, um die SSA-Form zu erhalten. Wir verbinden daher die beiden unterschiedlichen Semantiken für die Darstellung der intra- und interprozeduralen Datenflussabhängigkeiten mit einer neuen Operationsart, die wir *Filter* nennen. Wie wir oben erwähnt haben, entspricht sie in der intraprozeduralen Sicht einer *Proj*-Operation und hat die *Call*- bzw. *Start*-Operation als Vorgänger. In der interprozeduralen Sicht vereinigt sie die Datenflussvorgänger mit der Semantik einer *Phi*-Operation.

Eine *Phi*-Operation mit genau einem Vorgänger entspricht der Identität. Da die *Filter*-Operation in den beiden Sichten unterschiedliche Bedeutung hat, darf sie jedoch nicht entfernt oder übersprungen werden.

Aus der Semantik der *Filter*-Operation ergeben sich weitere Einschränkungen an ihren zugehörigen Grundblock. Interprozedural muss die Anzahl der Steuerflussvorgänger des Grundblocks mit der Anzahl der Datenflussvorgänger der *Filter*-Operation übereinstimmen. Intraprozedural hingegen muss der Vorgänger der *Filter*-Operation eindeutig sein, d. h. er befindet sich im selben Grundblock oder ist über alle Vorgänger-Grundblöcke eindeutig. Befindet




(a) Intraprozedurale Sicht

(b) Interprozedurale Sicht

Abbildung 3.1: Abhängigkeiten an der Aufrufstelle

sich der Datenflussvorgänger in einem direkt vorhergehenden Grundblock, so besitzt der Grundblock der *Filter*-Operation diesen als einzigen Vorgänger.

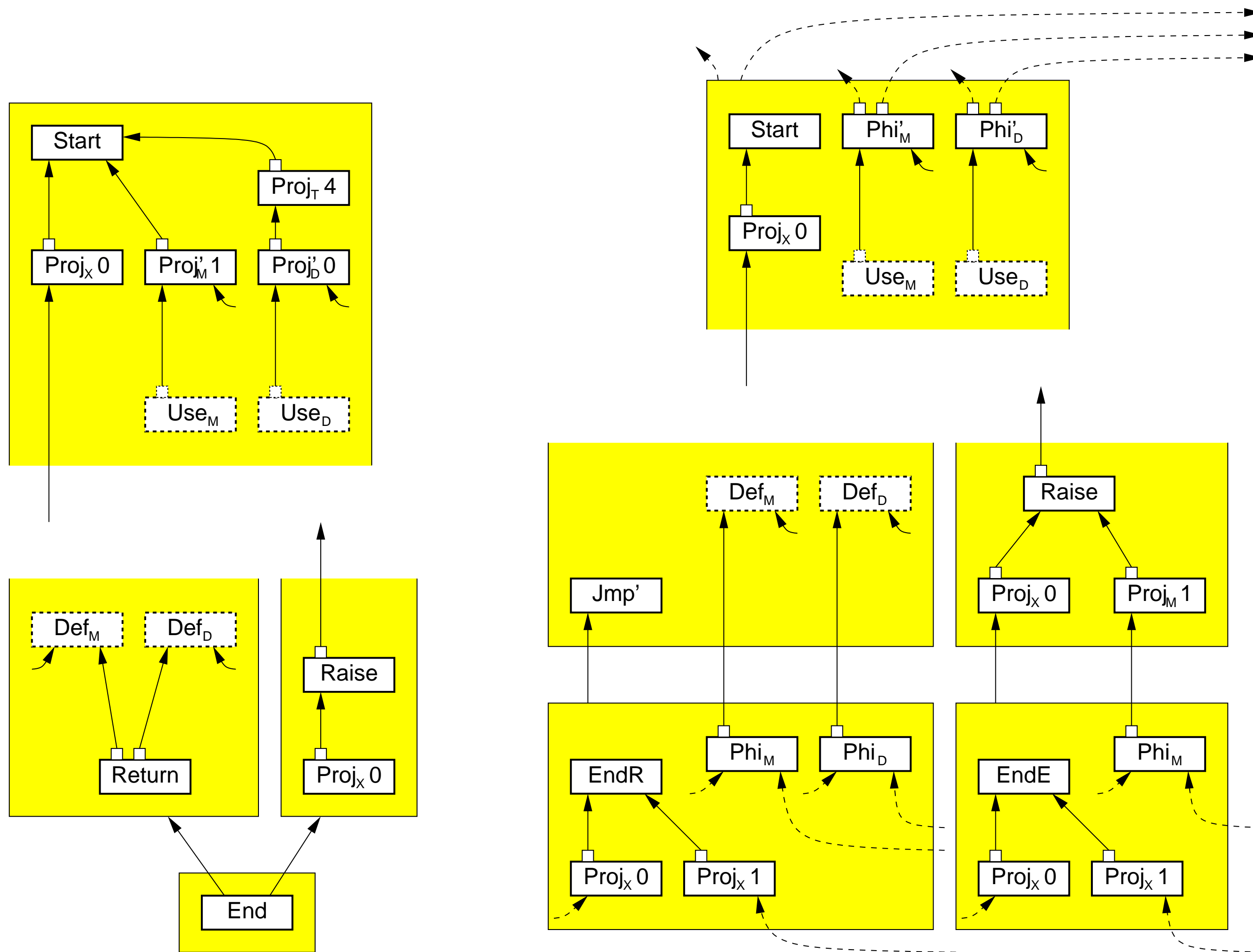
In Abschnitt 2.2 auf Seite 14 sind wir auf die notwendige Aufteilung des Grundblockes an einer Aufrufstelle eingegangen. Wir nehmen an, dass auch die *Filter*-Operationen in der intra- und interprozeduralen Sicht den selben Grundblöcken angehören. Dann verschärft sich diese Aufteilung jetzt folgendermaßen:

1. Der Grundblock, der intraprozedural dem Methodenaufruf folgt, hat intraprozedural genau ersten als Steuerflussvorgänger und interprozedural die *EndReg*-Operationen der Methoden, die von dieser Aufrufstelle aufgerufen werden können. Weiterhin befinden sich die *Filter*-Operationen für die Ergebniswerte und den Speicherzustand des Methodenaufrufes in dem Grundblock, der der Aufrufstelle intraprozedural folgt.

2. Die entsprechende Aussage gilt für den Grundblock, der die Ausnahmen eines Methodenaufrufes behandelt, sowie für die *Filter*-Operation für den Speicherzustand beim Abbruch der Methode durch das Werfen einer Ausnahme.

Abschließend stellen wir die Unterschiede der intra- und interprozeduralen Sicht exemplarisch an zwei Beispielen dar. Abbildung 3.1 zeigt eine Aufrufstelle, an der zwei unterschiedliche Methoden aufgerufen werden können. Die Grundblöcke sind dabei wie oben beschrieben aufgeteilt. Aufgrund der besseren Lesbarkeit sind die *Filter*-Operationen intraprozedural als *Proj'*- und interprozedural als *Phi'*-Operationen dargestellt. Die interprozeduralen Steuer- und Datenflussvorgänger sind rechts durch die gestrichelten Pfeile angedeutet.

In Abbildung 3.2 sind entsprechend die unterschiedlichen Abhängigkeiten am Anfang und Ende einer Methode dargestellt, die von zwei Programmstellen aufgerufen werden kann. Oben befindet sich jeweils der *Start*-Block mit den *Filter*-Operationen. Während intraprozedural der *End*-Block alle Operationen als Vorgänger besitzt, die die Methode beenden können, wird interprozedural die Methode durch zwei unterschiedliche Blöcke abgeschlossen. Um die Abbildung kompakt zu halten, ist jeweils nur eine *Return*- und *Raise*-Operation dargestellt, so dass die *Phi*-Operationen für die Rückgabewerte in diesem Fall überflüssig sind.




(a) Intraprozedurale Sicht

(b) Interprozedurale Sicht

Abbildung 3.2: Abhängigkeiten am Anfang und Ende von Methoden

## Unbekannte Vorgänger

Bisher haben wir nicht explizit berücksichtigt, dass an internen Aufrufstellen auch unbekannte Methoden bzw. dass interne Methoden auch von unbekannten Aufrufstellen aufgerufen werden können.

In den Sonderfällen, die wir in Abschnitt 2.4 besprochen haben, unterscheidet sich die intra- und interprozedurale Darstellung nicht. Die Abhängigkeiten der unbekannten Aufrufstellen bzw. Methoden sind dann implizit über die Semantik der Operationen dargestellt. Wir betrachten also nur Aufrufstellen, an denen sowohl explizite als auch unbekannte Methoden aufgerufen werden, und umgekehrt nur die Methoden, die sowohl von internen als auch von unbekannten Aufrufstellen aufgerufen werden können.

Da in der LIBFIRM-Bibliothek die Abhängigkeiten nur gegen den Steuer- bzw. Datenfluss explizit gespeichert werden, stellen wir die Abhängigkeiten von unbekannten Nachfolger-Operationen in der interprozeduralen Sicht überhaupt nicht dar. Es kann daher lebendige Operationen geben, die in der intraprozeduralen Sicht erreichbar sind, – im interprozeduralen Programmgraphen jedoch nicht.

Für die unbekannten Methoden bzw. Aufrufstellen haben wir keine expliziten Steuerfluss-Operationen *EndReg* und *EndExcept* bzw. *CallBegin* zur Verfügung. Wir fügen stattdessen eine neue Operation *Unknown* ein, und tragen diese zur Kennzeichnung von unbekannten Steuerflussvorgänger in die entsprechenden Grundblöcke ein. Andererseits stellen wir durch die gleiche Operation auch die unbekannten Datenflussvorgänger dar, und tragen sie entsprechend auch als Vorgänger in die *Filter*-Operationen ein.

Die *Unknown*-Operation verwenden wir also einerseits, um die Steuerflussvorgänger eines Grundblocks vollständig darstellen zu können. Im Datenfluss steht die *Unknown*-Operation aber für einen beliebigen, unbekannten Wert mit beliebigem Typ. Datenflussanalysen müssen diese Operation immer durch das Supremum der Analyse abschätzen.

Die Datenflussanalyse von *Filter*-Operationen mit unbekannten Vorgängern ist also nur dann sinnvoll, wenn aufgrund des Ausführungskontextes der *Filter*-Operation der unbekannte Vorgänger nicht berücksichtigt werden muss. Die hierfür benötigte Information ist einerseits eine bezüglich des Ausführungskontextes genauere Abschätzung der Funktionszeiger für den Methodenaufruf. Andererseits muss aus dem Ausführungskontext bestimmt werden, von welchen Aufrufstellen die Methode aufgerufen wird, die zur Ausführung der *Filter*-Operation führt.

## Aufwand

Wir betrachten den Aufwand für die Darstellung der interprozeduralen Abhängigkeiten, d. h. genauer den hierfür zusätzlich benötigten Speicher. Für jeden formalen Parameter der aufrufbaren Methoden, sowie für jeden Rückgabewert der Methoden-Aufrufstellen wird je eine *Filter*-Operation eingefügt. Außerdem werden für jede Methode und für jede Aufrufstelle zwei zusätzliche Grundblöcke und zugehörige Steuerfluss-Operationen eingefügt.

Wir nehmen an, dass die Anzahl der formalen Parameter und Rückgabewerte unabhängig von der Programmlänge sind. Dann verhält sich die Anzahl der eingefügten Operationen linear zur Summe der Anzahl von Methoden und Aufrufstellen, – insgesamt also linear zur Programmlänge.

Die Anzahl der eingefügten Steuer- und Datenflussabhängigkeiten verhält sich hingegen linear zur Größe der Aufrufrelation. In Abschnitt 2.2 auf Seite 15 haben wir beschrieben, dass diese im schlimmsten Fall quadratisch in der Programmlänge ist, – durch eine gröbere Abschätzung jedoch immer auf lineare Größe reduziert werden kann.

## Interprozedurale Datenflussanalysen

Wir gehen in diesem Abschnitt abschließend noch auf besondere Eigenschaften ein, die sich für interprozedurale Datenflussanalysen ergeben. Beim Durchlaufen der interprozeduralen Graphen ist es wichtig, die Methode der aktuell besuchten Operation zu kennen. Am Anfang einer Tiefensuche ist diese bekannt. Steuerflussvorgänger können sich nur in anderen Methoden befinden, wenn es sich dabei entweder um eine *CallBegin*-, eine *EndReg*- oder eine *EndExcept*-Operation handelt. In diesen Operationen speichern wir daher zusätzlich explizit ihre zugehörige Methode bzw. den zugehörigen Prozedurgraphen. Andererseits können nur die Datenflussvorgänger von *Filter*-Operationen anderen Methoden angehören. Da der Grundblock der *Filter*-Operation aber obige Operationen als Steuerflussvorgänger besitzt, kann man über diese auch auf die zugehörigen Methoden der Datenflussvorgänger schließen.

Wir betrachten weiterhin noch eine wichtige Eigenschaft, in der sich interprozedurale Abhängigkeitsgraphen von der intraprozeduralen SSA-Form unterscheiden. Die SSA-Form wird aufgebaut, in dem für jede statische Zuweisung an lokale Variablen Wertnummern eingeführt werden. Interprozedural müssen jedoch die lokalen Variablen aus unterschiedlichen Methodenausführungen – d. h. die unterschiedlichen Prozedurschachteln angehören –

unterschieden werden. Eine statische Definition kann daher zur Laufzeit für verschiedene Definitionen aus unterschiedlichen Methodenausführungen stehen. Wir betrachten dies an einem Beispiel genauer:

**Programm 3.1** Iterative und rekursive Berechnung von $f(n) := \sum_{i=1}^{n} i$ in der Programmiersprache C.

```
int sum_loop(int n) {
  int sum = 0;
  for (int i = 1; i <= n; ++i)
    sum = sum + i;
  return sum;
}

int sum_recursive(int n) {
  if (n == 0)
    return 0;
  else
    return sum_recursive(n - 1) + n;
}
```

Die Funktion `sum_recursive` berechnet die Summe rekursiv. Der interprozedurale Abhängigkeitsgraph besteht aus einer statisch festen, endlichen Anzahl von Operationen und Abhängigkeiten. Die Anzahl der gleichzeitig definierten Werte ist während der Laufzeit jedoch unbeschränkt. Bei der Ausführung der iterativen Variante `sum_loop` sind hingegen maximal soviele Werte gleichzeitig definiert, wie es Operationen in der Zwischendarstellung gibt.

Wir stellen also fest, dass in der interprozeduralen Darstellung die Abhängigkeiten auf die statischen Operationen dargestellt sind. Im Gegensatz zu intraprozeduralen Datenflussanalysen müssen jedoch in der Regel die dynamischen Definitionen einer statischen Operation durch den Ausführungskontext unterschieden werden.

## 3.2 Aufbau der Darstellung

In diesem Abschnitt skizzieren wir das Verfahren für den Aufbau der interprozeduralen Darstellung. Wir setzen dazu voraus, dass eine Abschätzung für die Aufrufrelation, wie wir sie in Abschnitt 2.1 auf Seite 13 definiert haben,

bereits vorliegt. Genauer heißt das, dass für jede Aufrufstelle die Menge der aufgerufenen Methoden eventuell inklusive der unbekannten Methode in der zugehörigen *Call*-Operation gespeichert ist. Weiterhin muss noch die Menge der Methoden gegeben sein, die von einer unbekannten Aufrufstelle aufgerufenen werden können. In Kapitel 4 werden wir eine einfache Analyse zur Bestimmung einer geeigneten Abschätzung angeben. Die Sonderfälle, die wir in Abschnitt 2.4 beschrieben haben, betrachten wir beim Aufbau der Darstellung nicht, da sie in der intra- und interprozeduralen Sicht übereinstimmen.

Der Aufbau lässt sich in zwei wesentliche Teile untergliedern. Im ersten Teil werden die Methoden für die Darstellung vorbereitet und Informationen zwischengespeichert. Im zweiten Teil werden für alle Aufrufstellen die Abhängigkeiten zwischen diesen und den aufgerufenen Methoden aufgebaut.

### Vorbereitung der Methoden

Für jede aufrufbare Methode fügen wir mit Grundblöcken die beiden künstlichen, interprozeduralen Steuerzusammenflüsse ein. Der erste Grundblock besitzt alle *Return*-Operationen der Methode als Vorgänger. Für jeden Rückgabewert der Methode sowie für den Speicherzustand wird jeweils eine *Phi*-Operation in diesen Grundblock eingefügt, die die unterschiedlichen Datenflussvorgänger der *Return*-Operationen zusammenfasst. Weiterhin fügen wir die interprozedurale Steuerfluss-Operation *EndReg* für den regulären Rücksprung aus der Methode ein.

Der andere Grundblock besitzt die Operationen als Vorgänger, die die Methode durch das Werfen einer Ausnahme beenden. In einer *Phi*-Operation werden die unterschiedlichen Speicherzustände nach Auslösen dieser Ausnahmen zusammengefasst. Durch die eingefügte *EndExcept*-Operation wird später der interprozedurale Steuerfluss dargestellt.

Da die beiden künstlichen Grundblöcke und ihre Operationen in der intraprozeduralen Darstellung nicht erreichbar sind, müssen wir uns diese für den gesamten Aufbau getrennt merken. Nachdem der Aufbau abgeschlossen ist, sind diese Operationen im interprozeduralen Programmgraphen erreichbar.

Wir betrachten wiederum zwei Spezialfälle, die besonders behandelt werden:

1. Es kann Methoden geben, die keine *Return*-Operationen besitzen, – beispielsweise Methoden, die nur durch Ausnahmen beendet werden. In diesem Fall wird der entsprechende Grundblock und die zugehörigen Operationen nicht erzeugt. In der interprozeduralen Sicht stellen wir die

Steuer- und Datenflussvorgänger stattdessen durch eine *Bad*-Operation dar, da dieser Steuerpfad offensichtlich nicht ausführbar ist.

2. Häufiger wird es hingegen Methoden geben, die nie eine Ausnahme werfen. Gibt es keine ausnahmewerfenden Operationen, so erzeugen wir analog den Grundblock und die beiden Operationen nicht, sondern stellen die Abhängigkeiten interprozedural ebenfalls durch eine *Bad*-Operation dar.

Weiterhin ersetzen wir die *Proj*-Operationen, die als Vorgänger die *Start*-Operation haben, durch *Filter*-Operationen. Wegen ihrer interprozeduralen Semantik verschieben wir diese noch gegebenenfalls in den *Start*-Block. Der *Start*-Block selbst wird in einen Block mit getrennten interprozeduralen Steuerflussvorgängern umgewandelt. Die Vorgänger werden erst beim Aufbau der Aufrufstellen eingetragen. Nur wenn die Methode auch von der unbekannten Aufrufstelle aufgerufenen werden kann, wird die *Unknown*-Operation unmittelbar als Steuer- und Datenflussvorgänger eingefügt.

## Aufbau der Aufrufstellen

Nachdem alle Methoden in dieser Weise vorbereitet und die nötigen Informationen zwischengespeichert sind, kann mit dem Aufbau der Aufrufstellen begonnen werden. Dafür teilen wir zuerst den Grundblock mit der *Call*-Operation in drei Grundblöcke auf: (a) ein Grundblock vor dem Methodenaufruf, (b) ein Grundblock nach dem Methodenaufruf und (c) ein Grundblock für die Behandlung der Ausnahmen durch den Methodenaufruf.

Der existierende Grundblock wird zu (b). Wir fügen einen neuen Grundblock (a) mit den ursprünglichen Steuerflussvorgängern ein, und verschieben alle *Phi*-Operationen aus (b) nach (a). Danach werden die *Call*-Operation und alle ihre transitiven Vorgänger in (b) ebenfalls nach (a) verschoben. Außerdem wird noch die interprozedurale Steuerfluss-Operation *CallBegin* mit dem Argument der *Call*-Operation eingefügt. In den Grundblock (a) wird ein expliziter Sprung eingefügt, der zum einzigen intraprozeduralen Steuerflussvorgänger von (b) wird. Die *Proj*-Operationen für die Ergebnisse des Methodenaufrufs ersetzen wir in (b) durch entsprechende *Filter*-Operationen. Die *Filter*-Operation für den Speicherzustand beim Wurf einer Ausnahme wird zusammen mit einem expliziten Sprung in den Grundblock (c) eingefügt.

Kann von der Aufrufstelle eine unbekannte Methode aufgerufen werden, so fügen wir die *Unknown*-Operation als interprozeduralen Steuerflussvorgänger

in die Grundblöcke (b) und (c), sowie als interprozeduralen Datenflussvorgänger in die *Filter*-Operationen ein.

Für jede weitere aufrufbare Methode müssen nun einerseits die interprozeduralen Abhängigkeiten vom Start der Methode auf die *CallBegin*-Operation sowie auf die Argumente der *Call*-Operation eingefügt werden. Andererseits müssen die Steuerflussabhängigkeiten der Grundblöcke (b) und (c) auf die *EndReg*- bzw. *EndExcept*-Operation und die Datenflussabhängigkeiten der *Filter*-Operationen auf die zwischengespeicherten Rückgabewerte der Methode dargestellt werden.

Der Aufbau für das gesamte Programm kann effizient durchgeführt werden. Der Aufwand für den Aufbau entspricht dem Speicheraufwand für die Darstellung (vgl. Seite 30) und ist linear zur Programmlänge bzw. zur Größe der Aufrufrelation.

## 3.3 Abbau der Darstellung

In der aufgebauten Darstellung existiert sowohl eine intra- als auch eine interprozedurale Sicht auf die Programmrepräsentation. Die intraprozedurale Sicht unterscheidet sich von der bisherigen Darstellung der Libfirm-Bibliothek nur an *Filter*-Operationen und der Einschränkung der möglichen Änderungen auf der Darstellung. Trotzdem kann es nötig sein, die Libfirm-Darstellung wiederherzustellen, da beispielsweise die Codeerzeugung diese voraussetzt.

Dieser *Abbau* erweist sich aber als sehr einfach. Die *Filter*-Operationen müssen wieder durch entsprechende *Proj*-Operationen ersetzt und die interprozeduralen Steuerflussvorgänger der Grundblöcke zurückgesetzt werden. Die erzwungene Aufteilung der Grundblöcke kann durch optimierende Transformationen einfach wieder rückgängig gemacht werden. Werden alle erreichbaren Knoten des intraprozeduralen Graphen anschließend noch in einen neuen Speicherbereich kopiert, so werden durch die Freigabe des alten Speicherbereichs insbesondere auch alle nicht mehr benötigten Operationen freigegeben.

# Kapitel 4

# Analyse der Aufrufrelation

In diesem Kapitel beschreiben wir eine Datenflussanalyse für die innere Abschätzung der Aufrufrelation, die wir in Abschnitt 2.1 auf Seite 13 definiert haben. Wir beschränken uns dabei auf eine intraprozedurale Analyse, da erst mit Hilfe des Analyseergebnisses die interprozedurale Darstellung aufgebaut wird. Ein Teil der Analyse lässt sich jedoch leicht auf die interprozedurale Sicht verallgemeinern, – wodurch in der Regel genauere Ergebnisse zu erwarten sind.

Wir machen allerdings eine starke Einschränkung an das zu übersetzende Programm. In der interprozeduralen Analyse von objekt-orientierten Programmen sind die polymorphen Methodenaufrufe von besonderem Interesse. Wird eine virtuelle Methode `f` an einem Objekt mit statischem Typ `T` aufgerufen, so können in der Regel an dieser Aufrufstelle alle Methoden aufgerufen werden, die die virtuelle Methode in einer Unterklasse von `T` überschreiben. Wenn Unterklassen in getrennten Übersetzungseinheiten definiert werden, können wir die an einer polymorphen Aufrufstelle aufrufbaren Methoden im allgemeinen nicht explizit abschätzen. Wir fordern daher, dass die vollständige Klassenhierarchie des Programms bekannt ist, und verbieten insbesondere das dynamische Laden von Klassen. Umgekehrt dürfen die getrennten Übersetzungseinheiten keine polymorphen Methoden aufrufen, sondern nur Methoden, die extern sichtbar sind, oder deren Funktionszeiger ihnen explizit bekannt ist.

Die Datenflussvorgänger der *Call*-Operation sind einerseits die Parameter für die aufgerufene Methode, und andererseits ein Funktionszeiger, der die aufzurufende Methode bestimmt. Die Unterscheidung dieser Vorgänger ist wichtig. Deshalb sprechen wir im ersten Fall von den Argumenten und im anderen Fall von dem *Funktionsargument* einer *Call*-Operation.

## 4.1 Freie Methoden

Die *Call*-Operation ruft eine Methode durch ihr Funktionsargument auf, – aber auch externe, unbekannte Aufrufstellen benötigen einen Funktionszeiger für den Aufruf einer Methode. In FIRM gibt es nur drei Möglichkeiten, den Funktionszeiger einer Methode zu bestimmen. Die *Const*-Operation speichert den Wert des Funktionszeigers, die *SymConst*-Operation kann den Funktionszeiger einer externen und einer extern sichtbaren, internen Methode zurückgeben, und die *Sel*-Operation gibt für eine virtuelle Methode und ein Objekt den Funktionszeiger für den dynamischen Typ zurück.

Wir müssen im folgenden mehrmals die Funktionszeiger bestimmen, die diese Operationen zur Laufzeit zurückgeben können. Da nicht entscheidbar ist, ob eine Operation überhaupt ausgeführt werden kann, ist auch die Menge der Funktionszeiger unentscheidbar. Stattdessen schätzen wir sie pessimistisch ab. Die Werte der *Const*- und *SymConst*-Operationen sind eindeutig. Die *Sel*-Operation schätzen wir im allgemeinen durch die Menge der Funktionszeiger ab, deren Methoden die angegebene virtuelle Methode überschreiben.[1] Ist der einzige Vorgänger einer *Sel*-Operation jedoch eine *Alloc*-Operation, so ist der dynamische Typ des Objekts bereits statisch bekannt und der zurückgegebene Funktionszeiger eindeutig.

Andererseits ist der Funktionszeiger einer *Sel*-Operation auch dann eindeutig, wenn es genau eine Implementierung der virtuellen Methode gibt. Wenn eine *Sel*-Operation allgemein nur einen Funktionszeiger zurückgeben kann, so wird sie unmittelbar durch eine entsprechende *Const*- oder *SymConst*-Operation ersetzt. Existiert hingegen weder eine interne noch eine externe Implementierung, so kann sie sogar durch eine *Bad*-Operation ersetzt werden.

Hat eine *Const*-, *SymConst*- oder eine *Sel*-Operation einen Funktionszeiger als Ergebnis, so ist sie in der Regel das Funktionsargument einer *Call*-Operation. Beispielsweise ist das für die Quellsprache JAVA [6] immer der Fall, da es dort keine expliziten Funktionszeiger gibt. In anderen imperativen Programmiersprachen können Funktionszeiger aber auch als Argumente von Methodenaufrufen übergeben (PASCAL [7]) oder sogar in Variablen gespeichert werden. In der Programmiersprache C [5] ist sogar Arithmetik auf Funktionszeigern definiert. Wir müssen daher auch die Fälle behandeln, in denen die oben genannten Operationen nicht Funktionsargumente einer *Call*-Operation sind.

Wir nennen einen Funktionszeiger und die zugehörige Methode *frei*, wenn

[1] Besitzt die virtuelle Methode eine Implementierung, so überschreibt sie sich auch selbst.

es einen Programmlauf und eine Operation gibt, so dass die Operation mit dem Funktionszeiger als Argument ausgeführt wird.[2] Die Menge der freien Methoden ist unentscheidbar. Deshalb bestimmen wir eine pessimistische Abschätzung, die wir in der Analyse der Aufrufrelation in zweifacher Hinsicht verwenden werden.

Da das Programmverhalten externer Methoden unbekannt ist, müssen wir alle externen Methoden, sowie alle extern sichtbaren, internen Methoden als frei annehmen. Wir gehen dann von allen statischen Operationen aus, die einen Funktionszeiger als Argument nehmen – mit Ausnahme dem Funktionsargument der *Call*-Operation. Ist die zugehörige Definition eine *Const*-, *SymConst*- oder *Sel*-Operation, so nehmen wir alle Funktionszeiger als frei an, die diese Definitionen zurückgeben können.

Dieses Verfahren lässt sich nur eingeschränkt auf die interprozedurale Sicht verallgemeinern, da im globalen Programmgraphen nicht alle ausführbaren Operationen erreichbar sind (vgl. 3.1 auf Seite 29). Für eine korrekte Abschätzung der freien Methoden müssen daher zusätzlich alle Methoden mit unbekannten Aufrufstellen intraprozedural analysiert werden.

## 4.2 Analyse

Wir bestimmen zunächst die Menge der internen Methoden mit externen Aufrufstellen. Diese besitzen in der inneren Abschätzung die unbekannte Aufrufstelle. Einerseits gehören die extern sichtbaren Methoden[3] zu dieser Menge. Andererseits können aber auch interne Methoden von externen Programmstellen aufgerufen werden, wenn ihr Funktionszeiger beispielsweise über den Speicher an externe Methoden übergeben wird. Dies ist allerdings nach Definition nur für die freien Methoden möglich. Deshalb schätzen wir die Menge der Methoden mit unbekannter Aufrufstelle durch die freien Methoden ab, deren Abschätzung insbesondere auch die extern sichtbaren Methoden enthält.

Anschließend schätzen wir für jede interne Aufrufstelle die Menge ihrer aufgerufenen Methoden ab, indem wir die Funktionsargumente rückwärts analysieren. Ist die Definition eines Funktionszeigers eine *Const*-, *SymConst*- oder *Sel*-Operation, so schätzen wir die Menge der Funktionszeiger wie oben ab. Ist der Vorgänger eine *Phi*-Operation, so bilden wir die Vereinigung über

[2] Dabei ist „Argument einer *Call*-Operation“ in obigem Sinne zu verstehen.

[3] Wir zählen auch die Methode des Hauptprogramms zu den „extern sichtbaren Methoden“

alle ihre Vorgänger. Ist der Datenflussvorgänger eine andere Operation, – z. B. eine *Load*-Operation – so können wir die Menge durch die freien Funktionszeiger abschätzen. In diesem Fall besitzt aber sowohl die Aufrufstelle eine unbekannte aufgerufene Methode als auch die Methode eine unbekannte Aufrufstelle. Dadurch werden die Analyseergebnisse zusätzlich eingeschränkt. Wir schätzen daher diese Operationen einfach durch die unbekannte Methode ab.

Aus dieser Abschätzung können monomorphe Methodenaufrufe bestimmt werden. Diese wurden aber bereits bei der Analyse der *Sel*-Operationen durch *Const*-Operationen ersetzt. Andererseits können unter Umständen auch Methoden bestimmt werden, die bei keiner Programmausführung aufgerufen werden. Wir bestimmen hierfür die Menge der *erreichbaren* Methoden. Dabei ist eine Methode genau dann erreichbar, wenn sie frei ist, oder wenn sie von einer erreichbaren Methode direkt aufgerufen werden kann. Die nicht erreichbaren Methoden können aus der Programmrepräsentation entfernt werden, und brauchen von nachfolgenden Optimierungen und der Codeerzeugung nicht berücksichtigt werden.

Abschließend betrachten wir noch eine Beispielumgebung mit der Quellsprache JAVA [6]. Wir nehmen an, dass von den internen Methoden nur das Hauptprogramm extern sichtbar ist. Externe Methoden sind monomorph und werden nur zur Ein-/Ausgabe verwendet. Insbesondere können sie keine internen Methoden aufrufen. Da für die Quellsprache JAVA [6] die Funktionsargumente von *Call*-Operationen immer *SymConst*-, *Const*- oder *Sel*-Operationen sind, gibt es auch keine freien Methoden. Ist das Hauptprogramm nicht rekursiv, so hat es nur die unbekannte Aufrufstelle. Alle anderen internen Methoden besitzen in der Abschätzung ausschließlich interne Aufrufstellen. An *Call*-Operationen können entweder nur unbekannte oder nur interne Methoden aufgerufen werden. Da die in Abschnitt 2.4 beschriebenen Fälle besonders behandelt werden, wird in dieser Umgebung die *Unknown*-Operation nicht benötigt. Weiterhin sind alle ausführbaren Operationen auch in der interprozeduralen Sicht erreichbar.

# Kapitel 5

# Zusammenfassung

In dieser Arbeit wird die Zwischensprache Firm erweitert. Interprozedurale Steuer- und Datenflussabhängigkeiten von Methodenaufrufen werden in einer neuen Sicht dargestellt. Dabei wird das Prinzip der Expliziten Abhängigkeitsgraphen und der SSA-Form auf interprozedurale Abhängigkeiten erweitert. Interprozedurale Datenflussanalysen können dadurch algorithmisch sehr ähnlich wie intraprozedurale Analysen formuliert werden. Da auch die interprozeduralen Abhängigkeiten explizit sind, können die Analysen einfach und effizient die über das gesamte Programm verteilten, benötigten Informationen erfassen.

Außerdem wurde eine einfache und effiziente Datenflussanalyse zur Abschätzung der Aufrufrelation vorgestellt. Diese wird verwendet, um den interprozeduralen Abhängigkeitsgraphen initial aufzubauen. Auf der interprozeduralen Sicht kann dann schrittweise die Abschätzung verfeinert und die Darstellung angepasst werden. Andererseits kann die Analyse selbst einige monomorphe Methodenaufrufe erkennen und nicht aufrufbare Methoden entfernen. Dies kann andere Optimierungen begünstigen und bereits zu wesentlich effizienteren Programmen führen.

## Ausblick

Die externen Methoden und Aufrufstellen wurden in dieser Arbeit nur sehr rudimentär behandelt, indem ihr Ausführungsverhalten stets als unbekannt angenommen wurde. An diesen Programmstellen wird die interprozedurale Datenflussanalyse auf dieser Darstellung stark eingeschränkt. In aggressiv optimierenden Übersetzern wird man daher versuchen, solche Situationen zu

vermeiden. Man spricht in diesem Zusammenhang auch von einer *geschlossenen* Übersetzung. Externe Methoden zur Ein-/Ausgabe wird man aber in der Regel trotzdem benötigen. Damit auch in diesen Fällen sinnvolle Analyseergebnisse über den globalen Speicher erzielt werden können, ist es insbesondere nötig, die Speicheränderungen dieser Methoden genauer abzuschätzen.

Zur Vereinfachung der interprozeduralen Abhängigkeiten wurden in jede Methode die beiden abschließenden Grundblöcke für das reguläre Ende und für den Abbruch der Methode durch eine Ausnahme eingefügt. Diese ersetzen interprozedural den bisherigen *End*-Block. Es ist aber vorteilhaft, diese Aufteilung generell in die intraprozedurale Darstellung einzuführen, da sich dadurch die Semantik des *End*-Blocks wesentlich vereinfacht. Weiterhin benötigt beispielsweise auch die Transformation für den offenen Einbau von Methoden die gleiche Darstellung.

Da noch keine kontextsensitiven interprozeduralen Datenflussanalysen für die in dieser Arbeit beschriebene Darstellung existieren, ist es praktisch nicht erwiesen, inwieweit sie für solche Analysen geeignet ist. Erst aufbauende Arbeiten werden zeigen, welche Vereinfachungen und Verbesserungen sich durch diese Darstellung ergeben.

# FIRM-Operationen

Wir beschreiben noch einmal ausführlich die Operationen, die für die zweisichtige Darstellung der Steuer- und Datenflussabhängigkeiten geändert oder eingefügt wurden.

### *Block*-Operation:

intraprozedural: $X_1 \times \ldots \times X_n \rightarrow BB$

interprozedural: $X_1 \times \ldots \times X_m \rightarrow BB$

Die erweiterte Operation unterscheidet zwei Zustände: der Grundblock hat (a) in beiden Sichten die selben Steuerflussvorgänger und (b) in beiden Sichten unterschiedliche Steuerflussvorgänger. Im Zustand (a) ist er mit der Operation der LIBFIRM [2] identisch. Im Zustand (b) besteht keine Beziehung zwischen den intra- und interprozeduralen Steuerflussvorgänger. Die interprozeduralen Steuerflussvorgänger befinden sich im allgemeinen nicht in der selben Methode wie die *Block*-Operation.

### *CallBegin*-Operation:

interprozedural: $P \rightarrow (X_1 \times \cdots \times X_n)$

Die Steuerflussoperation *CallBegin* ist nur in der interprozeduralen Sicht erreichbar und entspricht einer intraprozeduralen *Call*-Operation, auf die sie explizit verweist. Sie nimmt als einziges Argument einen Funktionszeiger auf die aufzurufende Methode. Die zugehörige *Call*-Operation speichert intern Referenzen auf $n$ Methoden. Dabei kann ein ausgezeichneter Eintrag auch für beliebige Methoden stehen. Stimmt der Funktionszeiger mit der $i$-ten Methode überein, so wird der Steuerfluss beim Nachfolger $X_i$ fortgesetzt. Das Verhalten ist undefiniert, wenn der Funktionszeiger mit keiner Methode übereinstimmt.

Die Parameter und Ergebniswerte werden nicht übergeben, da sie in der interprozeduralen Sicht explizit dargestellt sind. Da *CallBegin* eine interprozedurale Steuerfluss-Operation ist, speichert sie zusätzlich einen Verweis auf ihren Prozedurgraphen.

***EndReg*-Operation:** interprozedural: $. \rightarrow (X_1 \times \cdots \times X_n)$

Die Steuerflussoperation *EndReg* ist nur in der interprozeduralen Sicht erreichbar. Sie steht für den regulären Rücksprung einer Methode und fasst die *Return*-Operationen der intraprozeduralen Darstellung zusammen. Wird die Methode durch den $i$-ten Steuerflussvorgänger des zugehörigen *Start*-Blocks aufgerufen, so wird nach Ausführung der *EndReg*-Operation der Steuerfluss beim Nachfolger $X_i$ fortgesetzt.

Da die Nachfolger-Operationen sich in der Regel in anderen Methoden befinden, speichert die *EndReg*-Operation zusätzlich einen Verweis auf ihren Prozedurgraphen.

***EndExcept*-Operation:** interprozedural: $. \rightarrow (X_1 \times \cdots \times X_n)$

Die *EndExcept*-Operation ist nur in der interprozeduralen Sicht erreichbar. Analog zur *EndReg*-Operation fasst ihr Grundblock die Operationen zusammen, die die Methode durch das Werfen einer Ausnahme beenden können. Wird die Methode durch den $i$-ten Steuerflussvorgänger des *Start*-Blocks aufgerufen, so wird nach Ausführung der *EndExcept*-Operation der Steuerfluss beim Nachfolger $X_i$ fortgesetzt. Sie speichert ebenfalls einen expliziten Verweis auf ihren Prozedurgraphen.

***Filter*-Operation:**
intraprozedural: $T \rightarrow U$
interprozedural: $U_1 \times \ldots \times U_n \rightarrow U$

Die *Filter*-Operation hat intraprozedural die Semantik einer *Proj*-Operation und interprozedural die Semantik einer *Phi*-Operation. Die Anzahl der interprozeduralen Datenflussvorgänger muss mit der Anzahl der Steuerflussvorgänger des zugehörigen Grundblocks in der interprozeduralen Sicht übereinstimmen. In der intraprozeduralen Sicht muss die entsprechende Definition hingegen über alle Steuerflussvorgänger eindeutig sein. In der Regel befindet sie sich entweder im selben Grundblock wie die *Filter*-Operation oder in ihrem einzigen Grundblock-Vorgänger.

***Unknown*-Operation:** interprozedural: $. \rightarrow U$

Die *Unknown*-Operation steht einerseits für beliebige unbekannte Datenflusswerte mit beliebigem Typ. Datenflussanalysen müssen ihr Verhalten immer durch das Supremum der Analyse abschätzen. Andererseits kann die Operation auch zur Kennzeichnung unbekannter Steuerflussvorgänger verwendet werden. Diese müssen nicht erreichbar sein.

# Literaturverzeichnis

[1] M. Trapp, G. Lindenmaier und B. Boesler: *Documentation of the Intermediate Representation FIRM*, Tech. Bericht 1999-14, Fakultät für Informatik, Universität Karlsruhe (TH), Dezember 1999.

[2] G. Lindenmaier: *Tutorial for the Firm library*, Tech. Bericht, Fakultät für Informatik, Universität Karlsruhe (TH), September 2001.

[3] M. Trapp: *Optimierung objektorientierter Programme*, Springer Verlag, 2001.

[4] M. Armbruster und Ch. von Roques: *Entwurf und Realisierung eines Sather-K Übersetzers*, Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Dezember 1996.

[5] ISO/IEC 9899:1999: *Programming languages – C*, 2. Ausgabe, Dezember 1999.

[6] J. Gosling, B. Joy, G. Steele und G. Bracha: *The Java(tm) Language Specification*, Addison Wesley, 2. Ausgabe, 2000.

[7] K. Jensen und N. Wirth, *PASCAL: User Manual and Report*, Springer Verlag, 1974.