Datenflußanalyse in objekt-orientierten Programmiersprachen - Von der Theorie zur Praxis

Habilitationsschrift

vorgelegt dem

Fachbereich Elektrotechnik und Informatik

der

Universität-Gesamthochschule Siegen

am

23. April 1997

von

Dr. rer. nat. Wolfgang Golubski

Gelsenkirchen

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Hintergrund und Motivation

Objekt-orientierte Programmiersprachen ausgestattet mit Vererbungsmechanismen unterstützen vor allem schnelle Prototypentwicklung von Programmen und haben einen hohen Grad an Wiederverwendbarkeit von existierenden Ressourcen. Aber eine breite Akzeptanz bei den Benutzern von Programmiersprachen bleibt ihnen weiterhin verwehrt, obwohl sich objekt–orientierte Ansätze im Bereich der Softwareentwicklung, z.B. OO-Analyse nach Booch, oder im Bereich der Datenbanken immer mehr durchsetzen.

Einen kritischen Punkt der objekt–orientierten Programmiersprachen stellt ihre Implementierung dar. Abgesehen von erheblichem Speicherplatzbedarf eines objekt–orientierten Systems kämpfen Programme dieser Sprachen im Gegensatz zu gleichbedeutenden C/C++-Programmen mit ihrer „problematischen" Laufzeiteffizienz. Daher spielen Optimierungstechniken für objekt–orientierte Sprachen zur Steigerung der Effizienz eine übergeordnete Rolle.

Die vorliegende Arbeit beschäftigt sich mit einem Modell zur statischen Analyse von ungetypten objekt–orientierten Programmen. Die behandelte Programmiersprache ist eine Smalltalk-80–artige Sprache.

Im Vergleich zu imperativen Programmiersprachen werden in objekt–orientierten Programmen weitaus häufiger Methodenaufrufe als Funktions- oder Prozeduraufrufe in imperativen Sprachen verwendet. Die Implementierungstechniken für imperative Sprachen sind soweit entwickelt, daß sich die Kosten eines Funktionsaufrufes im akzeptablen Rahmen befinden. Hier sei an die C/C++-Implementierungen erinnert, die mittels exzessiver Inline-Technik die von Funktionsaufrufen verursachten Kosten

(zusätzlicher Aufwand durch Anlegen der Umgebung, des Activation Records) minimieren. In imperativen Sprachen können Funktionsaufrufe statisch an ihre Funktionsdefinition gebunden werden. Und dies ist vor allem in ungetypten dynamisch gebundenen objekt–orientierten Sprachen nicht ohne weiteres möglich, da erst zur Laufzeit durch den Empfänger eines Aufrufes die entsprechende Methodendefinition bestimmt wird. Der Empfänger ist ein ausgezeichnetes Argument eines Aufrufs und kann nicht unmittelbar anhand des Programmtextes exakt ermittelt werden. In getypten objekt–orientierten Sprachen kann aufgrund der Typdeklarationen der Empfänger wesentlich genauer erfaßt werden, sodaß das soeben angesprochene Problem in Sprachen wie C++ oder Eiffel nicht so relevant ist.

1.1 Implementierung von Methodenaufrufen

Wir betrachten einen Methodenaufruf in Smalltalk-80 [GR83] und beschreiben in welcher Form zu einer Aufrufstelle der geeignete Methodencode (-definition) ermittelt werden kann.

Sei `revr sela:v selb:w` ein Methodenaufruf mit Empfänger `revr` und Argumenten `v` und `w`.

Abbildung 1.1: Naive Implementierung der Methodenaufrufe

In einer naiven Implementierung, siehe Abbildung 1.1, übersetzt der Compiler den Methodenaufruf in einen Zwischencode, z.B. der Form *call sela:selb:(revr,v,w)*. Wenn dieser Zwischencode vom Laufzeitsystem (Interpreter) interpretiert wird, so wird zunächst eine systemeigene Methodensuchroutine (*method lookup routine*) ausgeführt. Die Routine schaut in der Klasse nach, zu der das Empfängerobjekt gehört, ob ein geeigneter Methodencode vorhanden ist. Wenn dies der Fall ist, dann kann die Verbindung mit der Aufrufstelle (Anlegen der Umgebung, etc.) hergestellt werden. Ansonsten wird in der Superklasse der gerade untersuchten Klasse nach der Methode weitergesucht. Dieser Prozeß wird solange durchgeführt, bis eine geeigne-

te Methode gefunden wird oder es keine Superklasse mehr gibt. Letzteres führt zu einem Programmabbruch mit der Fehlermeldung *does–not–understand*.

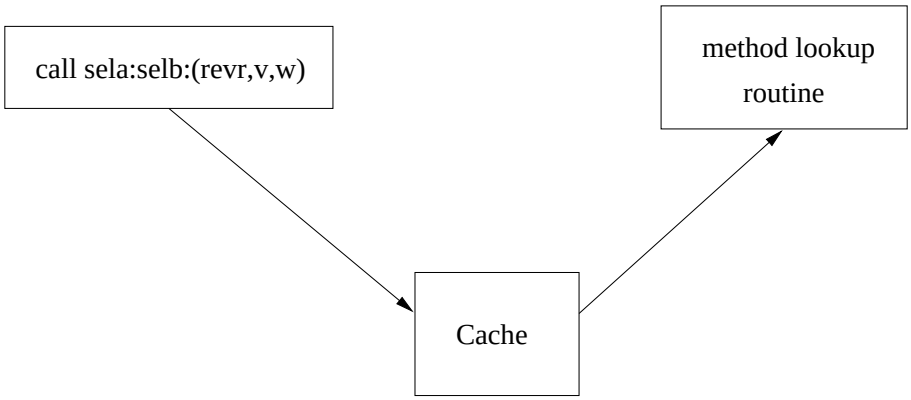


Abbildung 1.2: Deutsch/Schiffman Implementierung der Methodenaufrufe

Eine verbesserte Methodenaufruf-Implementierung, siehe Abbildung 1.2, die schon von Deutsch und Schiffman [DS84] vorgeschlagen wurde, wird auch heute noch von manchen Implementierungen verwendet. Hierbei wird ein zusätzlicher globaler Cache eingeführt, auf den bei jedem Methodenaufruf zunächst zugegriffen wird. Der Cache speichert die letzten n unterschiedlichen Methodenaufrufe, genauer die Zeiger auf den entsprechenden Methodencode, ab. Er wird als Hashtabelle angelegt, wobei in jedem Eintrag die Klassenzugehörigkeit des Empfängers, der Methodenname und der Zeiger auf den entsprechenden Methodencode verwaltet werden. Wenn im Cache keine geeignete Methode gefunden wird, so kommt die Methodensuchroutine wieder zur Ausführung.

Abbildung 1.3: Direkter Zugriff auf den Methodencode

Wünschenswert wäre eine Implementierung der Methodenaufrufe in der Art, daß bereits der Compiler einen direkten Zugriff auf den Methodencode ermöglichen könnte, siehe Abbildung 1.3. Dies kann in einer dynamisch gebundenen Programmiersprache i.a. natürlich nicht der Fall sein, aber der Compiler kann versuchen, möglichst viele Aufrufstellen statisch zu binden.

1.2 Compilierung in objekt–orientierten Systemen

Objekt–orientierte Systeme, wie z.B. Smalltalk-80, beinhalten neben der Programmierprache und einer reichhaltigen Programmbibliothek (Klassenbibliothek) auch eine graphisch unterstützte Programmierumgebung, in der Browser, Editor, Compiler und Debugger integriert sind. Solche Systeme eignen sich hervorragend zur schnellen Prototypentwicklung. Der herkömmliche Begriff eines Programmes existiert dort nicht. Stattdessen wird ein Ausdruck bzgl. der vollständigen Klassenbibliothek evaluiert. Methodendefinitionen werden direkt nach ihrer Eingabe in Zwichencode übersetzt. Somit ist eine den "klassischen" Pascal- oder Modula–Programmen vergleichbare vollständige Compilierung eines Programmes nicht möglich. Dadurch sind manche weitreichenden Optimierungstechniken ausgeschlossen. Nach der Prototypentwicklung gilt es, ein effizientes Programm zu erstellen. Dies kann durch Neuentwicklung in einer effizienten Programmiersprache, wie z.B. C/C++, erfolgen oder durch Zusammenstellung eines Programmes aus der übergroßen Klassenbibliothek und seiner Compilierung unter Zuhilfenahme von statischen Programmanalysen.

In dieser Arbeit möchten wir einen Schritt von der Prototypentwicklung in Richtung Endproduktentwicklung gehen und werden ein Konzept für statische Programmanalysen vorstellen, das für die Compilation eines gesamten Programmes genutzt werden kann.

1.3 Statische Programmanalysen

Die Aufgabe der statischen Programmanalyse besteht in der statischen Gewinnung von Informationen über das Laufzeitverhalten von Programmen. Der Nutzen von statischer Programmanalyse kann durch die folgende Frage ausgedrückt werden:

> *Gibt es zu einem gegebenen Programm ein semantisch äquivalentes Programm, das weniger Zeit und/oder Platz benötigt als das Ursprungsprogramm ?*

Fragen dieser Art sind i.a. unentscheidbar, d.h. es gibt keinen Algorithmus, der diese Frage korrekt und vollständig beantwortet. In der Praxis häufig verwendete Programmanalysen beschäftigen sich mit den Fragen

- Hat ein Ausdruck an einem Programmpunkt immer den gleichen (konstanten) Wert? Wenn ja, so kann bereits zur Übersetzungszeit der Wert berechnet und das Ergebnis im Programm eingesetzt werden. (Constant Propagation Problem)

- Wenn ein Ausdruck mehrfach im Programm vorkommt, ergibt seine Berechnung an mindestens zwei Programmpunkten immer den gleichen Wert? Wenn ja, so ist die zweite Berechnung überflüssig und kann durch einen geeigneten Verweis auf das Ergebnis des Ausdrucks ersetzt werden. (Code Motion Problem)

- Ist der Empfänger einer Methode immer derselbe? Wenn ja, so kann eine statische Bindung des Methodenaufrufes erfolgen. (Type Analysis Problem)

Das Einsatzgebiet der statischen Programmanalyse geht über die Anwendung innerhalb eines Compilers zur Optimierung hinaus. So können Informationen über Benutzung von Variablen zur Unterstützung von Entwicklung, Test und Dokumentation von Programmen sowie ihrer Verifikation dienen.

Wichtige Gesichtspunkte statischer Programmanalysen sind ihre garantierte Terminierung, die Gewährleistung ihrer Korrekheit und eine akzeptable Laufzeit sowie ein akzeptabler Speicherbedarf.

Statische Programmanlysen können nicht für alle Eingabedaten eines Programmes ausgeführt werden, weshalb Vereinfachungen bzw. Abstrahierungen vorgenommen werden müssen. Es werden die vom Programm verwendeten (konkreten) Daten ersetzt durch (abstrakte) Daten, die auf die spezielle Fragestellung der Analyse abgestimmt sind. Ausgehend von einer dynamischen Semantik einer Programmiersprache läßt sich folgender Zusammenhang zur Programmanalyse herstellen: Die semantischen Funktionen einer dynamischen Semantikdefinition sind i.a. nicht berechenbar. Da eine Programmanalyse ihre Informationen in endlicher Zeit berechnen muß, werden die semantischen Funktionen derart modifiziert, daß sie berechenbar sind. Die so gewonnene Semantik wird als Nichtstandard-Semantik bezeichnet. Die Berechnung der Nichtstandard–Semantik liefert Informationen über die Standard–Semantik (dynamischen Semantik) eines Programmes. Es gibt eine Abstraktionsrelation, die die beiden Semantiken miteinander verbindet, siehe Abbildung 1.4.

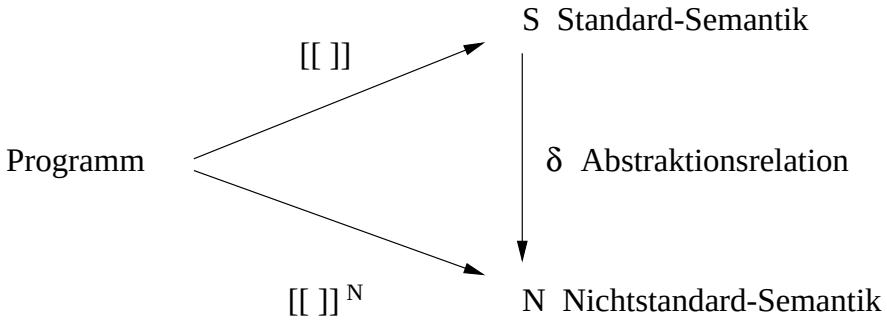


Abbildung 1.4: Abstraktion der Standard-Semantik

Umgekehrt können durch eine sogenannte Konkretisierungsrelation, siehe Abbildung 1.5, die die abstrakten Informationen in die Menge der konkreten Daten abbildet, angegeben werden.[1]

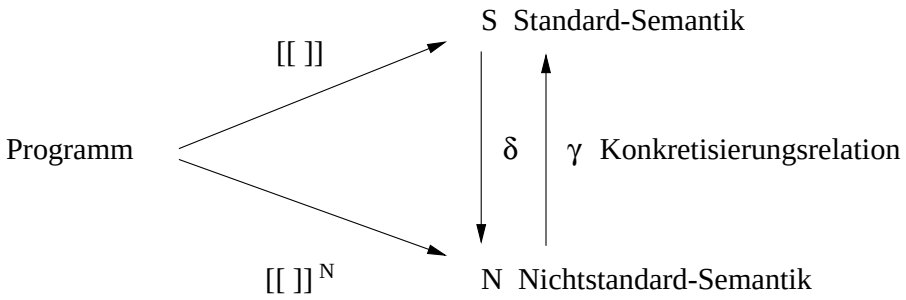


Abbildung 1.5: Konkretisierung der Nichtstandard-Semantik

In der vorliegenden Arbeit werden wir die Korrektheit der Nichtstandard–Semantik bzgl. einer Standard–Semantik nicht behandeln, sondern werden eine Nichtstandard–Semantik definieren, die das spezifische Problem beschreibt, aber immer noch nicht berechenbar ist. Wir zeigen dann, daß eine algorithmische Beschreibung der Nichtstandard–Semantik existiert und korrekt ist.

Somit ist die Nichtstandard–Semantik N^R die Referenzlösung unserer Programmanalyse und N^A die Lösung der algorithmischen Umsetzung, siehe Abbildung 1.6. Abhängig von der mathematichen Struktur der Informationen erhalten wir die Gleichheit (Koinzidenz) oder das Enthalten sein (die Korrektheit) der algorithmischen Lösung mit bzw. in der Referenzlösung.

[1]Diese Eigenschaft, die zur Konstruktion von "konkreten" abstrakten Interpretationen verwendet wird, ist von Cousot und Cousot in [CC77, CC79] beschrieben worden. Eine systematische Übersicht des abstrakten Interpretationsansatzes ist in [CC96] zu finden.

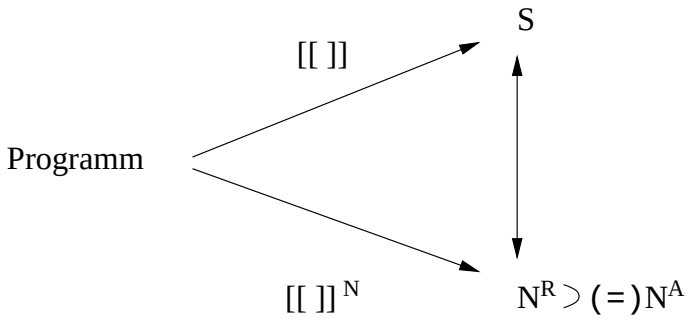


Abbildung 1.6: Korrektheit der algorithmischen Lösung der Nichtstandard-Semantik

In dieser Arbeit werden statische Programmanalysen als globale Datenflußanalysen dargestellt, insbesondere für ungetypte objekt–orientierte Programmiersprachen. Bei Datenflußanalysen unterscheidet man zwischen intra- und interprozeduralen Analysen, je nachdem ob nur eine einzelne Prozedur isoliert betrachtet wird oder der durch Prozeduraufrufe entstehende Datenfluß erfaßt wird. Eine ausführliche Einführung in dieses Themengebiet kann in [Zi82, MJ81, WM95] gefunden werden.

Um ein Programm möglichst gut optimieren zu können, werden verschiedene Datenflußanalysen eingesetzt. Für diese Analysen müßte jedesmal ein Algorithmus entworfen und seine Korrektheit gezeigt werden. Diesen Aufwand können wir minimieren, da die Struktur der Algorithmen und der Korrektheitsbeweise in wesentlichen Teilen gleich sind. Daher kann ein sogenannter Datenflußanalyse–Rahmen eingeführt werden, wie der für imperative Sprachen von Knoop, Steffen und Rüthing in [Kn93, KRS94c]. Weitere Datenflußanalyse-Rahmen werden in [CDG96b] diskutiert. Für ein konkretes Datenflußanalyseproblem genügt es, den Datenbereich und gewisse Operationen auf ihm zu spezifizieren. Der Datenflußanlayse–Rahmen stellt einen Algorithmus bereit, der das spezielle Datenflußanalyseproblem löst. Der Datenflußanalyse–Rahmen von Knoop, Steffen und Rüthing wird an objekt–orientierte Sprachen angepaßt, siehe auch [KG96a, KG96b, KG96b, Go96, Go97, GP97b], so daß über diese Arbeit hinausgehend auch andere Datenflußanalyseprobleme gelöst werden können.

1.4 Typanalyse

Ausgangspunkt unserer Überlegungen war die Fragestellung, ob zur Übersetzungszeit der Empfänger eines Methodenaufrufes eindeutig einer Klasse zugeordnet wer-

den kann. Diese Klassenzugehörigkeit werden wir von nun an mit Hilfe des Begriffs *Typ* ausdrücken. In Variablen werden Werte (Objekte) abgespeichert. Ein Objekt gehört zur Laufzeit eindeutig einer Klasse an, d.h. ist Instanz einer Klasse. Der Typ einer Variablen zur Laufzeit ist genau diese Klasse bzw. der Klassenname. In der statischen Programmanalyse sind Typen Invarianten bzgl. der Laufzeit eines Programmes. Die statische Auswertung von Verzweigungen kann dazu führen, daß an einem Programmpunkt eine Variable sowohl Instanz einer Klasse A als auch Instanz einer Klasse B sein könnte. Daher ist der Typ einer Variablen die kleinste Menge der möglichen Klassennamen, von denen die Variable eine Instanz gespeichert haben könnte.

$$Typ \; = \; Menge\ von\ Klassen(namen)$$

Nicht nur Variablen werden Typen zugeordnet, sondern auch Ausdrücken. Die oben angesprochenen Invarianten beinhalten, daß

- in einer Variablen an einem Programmpunkt nur Werte ihres Typs abgelegt werden können,
- die Berechnung eines Ausdrucks nur Werte seines Typs liefern kann.

1.5 Typanalyse und Datenflußanalyse

Wir werden in dieser Arbeit im angesprochenen Datenflußanalyse–Rahmen eine Programmanalyse definieren, die den Typ für jeden Empfänger eines Methodenaufrufes bestimmt. Ist dieser Typ eindeutig, d.h. die Menge der Klassen dieses Typs besteht nur aus einem Element, so kann der Methodenaufruf wie ein Funktionsaufruf behandelt werden.

Eine "vernünftige" Typanalyse im Modell der klassischen interprozeduralen Datenflußanalyse zu definieren, ist nicht ohne weiteres durchführbar. Die klassische interprozedurale Datenflußanalyse konstruiert zunächst einen Kontrollflußgraphen, der die Aufrufstruktur der Prozeduren des Programmes wiederspiegelt. Dann wird der Kontrollflußgraph analysiert. In objekt–orientierten Sprachen führt dieser Ansatz zu folgender Problematik:

- die Konstruktion des Kontrollflußgraphen hängt von den Typinformationen jeder einzelnen Aufrufstelle ab,
- die Berechnung der Typinformationen hängt von der Analyse des Kontrollflußgraphen ab.

Daher haben wir von einer "vernünftigen" Typanalyse gesprochen, weil natürlich ein Kontrollflußgraph aufgebaut werden kann, der jede Aufrufstelle mit allen Methodendefinitionen geeigneten Namens verbindet. Die Analyse eines solchen Graphens liefert aber i.a. sehr ungenaue Informationen, weil ja nicht jede Methodendefinition zur Laufzeit ausgeführt wird. Um die gegenseitige Abhängigkeit von Kontrollflußgraph und Typinformation aufzulösen, wird folgende Vorgehensweise gewählt:

- Aufbau eines Kontrollflußgraphen, indem jede Aufrufstelle mit allen Methodendefinitionen geeigneten Namens verbunden wird,
- zur Analysezeit wird selektiert, welche Methodendefinitionen analysiert werden. Dabei werden von der Analyse bereits berechnete Typinformationen verwendet.

Typanalysen für objekt-orientierte Programme sind seit längerem bekannt. Aber erst in den letzten drei Jahren sind aggressivere Typanalysen vorgestellt und getestet worden. In dieser Arbeit werden wir fünf Typanalysen unterschiedlicher Aggressivität im Rahmen der interprozeduralen Datenflußanalyse vorstellen und neben theoretischen Vergleichen auch, und vor allem, empirische Aspekte untersuchen. Die Frage nach den Grenzen (dem Trade–Off) von Theorie und Praxis werden ausführlich diskutiert.

1.6 Graphische Darstellung der Ergebnisse der Typanalyse

Neben der Optimierung der Implementierung von objekt-orientierten Programmen liegt der Einsatzbereich einer Typanalyse in der Unterstützung der Entwicklung von Programmen. Dazu eignen sich in hervorragender Weise graphische Werkzeuge. Ein ungetyptes Programm kann nach erfolgter Typanalyse mit Typinformationen

versehen (annotiert) werden. Dies erhöht die Lesbarkeit und das Verständnis des Programms. Ebenso können mögliche Fehlerquellen einfacher lokalisiert werden. Die von einer Typanalyse berechneten Informationen sind bei größeren Programmen sehr umfangreich, so daß eine graphische Aufbereitung notwendig ist. Aus diesen Gründen haben wir ein Werkzeug namens SOLAT entwickelt [Po97, GP97b], daß genau diese Aufgabe erfüllt.

1.7 Ziele der vorliegenden Arbeit

Im Rahmen dieser Arbeit wurden folgende Ziele verfolgt und umgesetzt:

- Bereitstellung eines interprozeduralen Datenflußanalyse–Rahmens für objekt–orientierte Programme, der Korrektheitsbeweise der speziellen Datenflußanalyseprobleme erleichert,
- Typanalysen für ungetypte objekt–orientierte Sprachen,
- Theoretischer Vergleich verschiedener Typanalysen bzgl. ihrer Mächtigkeiten,
- Implementierung der Typanalysen,
- Empirischer Vergleich der Typanalysen bzgl. verschiedener Aspekte (Laufzeit, Qualität der gewonnenen Informationen, Speicherplatz)
- Bereitstellung eines graphischen Werkzeugs zur Handhabung der Typinformationen.

1.8 Aufbau der vorliegenden Arbeit

Nach diesem einführenden Teil der Arbeit wird in Kapitel 2 eine einfache objekt–orientierte Sprache namens SOL vorgestellt. Durch Transformation der SOL -Programme in Graphen wird die Grundlage für eine Datenflußanalyse geschaffen. In Kapitel 3 wird der Datenflußanalyse-Rahmen beschrieben und die Korrektheit des algorithmischen Ansatzes bzgl. seiner Referenzlösung gezeigt. Danach werden in Kapitel 4 fünf verschiedene Typanalysen definiert und ihre Korrektheit gezeigt. Kapitel

5 beschäftigt sich mit den empirischen Untersuchungen und ihrer Resultate. Dabei wird zunächst eine Neuformulierung des theoretischen Algorithmuses erfolgen, sodaß nur noch diejenigen Teile des Programms ausgewertet werden, die unbedingt benötigt werden. Danach werden die fünf Typanalysen bzgl. ihrer Analysezeiten, ihrer Qualität (Anzahl der mehrdeutigen Aufrufstellen, Anzahl der überflüssigen Methodendefinitionen, Speicherplatzbedarf) miteinander verglichen. Die einzelnen Aspekte werden zusammenfassend bewertet, wobei sich eine Typanalyse als der beste Kompromiß zwischen Theorie und Praxis herauskristallisiert. In Kapitel 6 wird ein Werkzeug zur graphischen Darstellung der berechneten Typinformationen beschrieben. Verwandte Ansätze (Kapitel 7) skizzieren andere Konzepte zur Berechnung von Typinformationen und vergleicht diese mit dem hier beschriebenen Verfahren. In den Schlußbemerkungen des Kapitels 8 wird neben einer Zusammenfassung der Arbeit ein Ausblick auf nicht behandelte Fragestellungen gegeben.

Kapitel 2

Eine objekt-orientierte Programmiersprache

Zunächst werden wir eine Smalltalk-80–ähnliche Sprache vorstellen und definieren. Einige Programmtransformationen werden dann zu der für die Datenflußanalyse geeigneten Programmdarstellung führen.

2.1 Die Sprache SOL

Ein SOL -Programm besteht aus einer Menge von Klassendefinitionen (der Klassenbibliothek) und einem Hauptprogramm (einem Ausdruck). Dieser Ausdruck wird bezüglich der Klassenbibliothek ausgewertet, d.h. unter Zuhilfenahme aller durch die Klassenbibliothek zur Verfügung gestellten Ressourcen. Jede Auswertung eines Ausdrucks liefert einen Wert zurück. Dies kann entweder eine Integerzahl, der undefinierte Wert `nil` oder ein Zeiger auf ein Objekt sein. Ein Objekt ist eine Instanz einer Klasse. Es besteht aus seinen Instanzvariablen und einem Hinweis auf die Klasse, die es erzeugte. Eine Klasse definiert Instanzvariablen und Methoden.

Die sogleich definierte Sprache SOL stellt folgende Sprachkonstrukte zur Verfügung:

- Zuweisungen (*assignments*),
- Erzeugung eines Objektes durch Klasseninstanziierung,
- dynamisches Binden von Methodenaufrufen,

- *if-then-else-, while-, repeat*-Konstrukte, wie von imperativen Sprachen her bekannt.

Die wesentlichen Unterschiede zur Smalltalk-80-Sprache sind gekennzeichnet durch den Verzicht auf:

- Vererbung: Das statische Vererbungskonzept von Smalltalk-80 läßt sich durch eine Programmtransformation auflösen. Dies geschieht, indem jede Klassendefinition erweitert wird um ihre ererbten Variablen- und Methodendefinitionen mit Ausnahme der überladenen, d.h. überdefinierten (overloading), Methoden. Zusätzlich kann jede super–Botschaft ersetzt werden durch eine self–Botschaft an eine umbenannte Version der entsprechenden ererbten Methode. Somit kann in dem transformierten Programm jede Klasse als separate Einheit betrachtet werden, da sie ja alle ererbten Definitionen enthält.[1]

- Blockkonzept von Smalltalk-80: Dieses Konzept läßt sich ebenfalls durch eine geeignete syntaktische Transformation in der Sprache SOL simulieren[2].

- I/O-Routinen, File–Handling und eine Reihe von primitiven Werten (Floats, Characters,...).

2.2 Die Syntax von SOL

Ein Programm besteht aus einer Menge von Klassen und einem Ausdruck, siehe Abbildung 2.1. Eine Klasse setzt sich aus ihrem Namen (*ClassId*), der Definition von Instanzvariablen (*Id*) und einer Menge von Methodendefinitionen (*Method*) zusammen. Eine Methodendefinition (*Method*) besteht aus dem Methodennamen

[1] Vererbung ist ein wichtiges Konzept zur Wiederverwendung von bereits definierten Ressourcen. Allerdings im Rahmen der statischen Programmanalyse würde die Einbeziehung von Vererbung nur zu einer unübersichtlicheren Darstellung führen, sodaß wir auf sie verzichtet haben.

[2] In Smalltalk-80 werden alle Kontrollstrukturen durch Blöcke programmiert, die Ausdrücke darstellen, deren Auswertung verzögert werden kann. Eine Blockauswertung kann somit als eine anonyme Methodenausführung betrachtet werden. Genauer, die Definition eines Blockes kann durch eine Instanziierung einer neuen Klasse und die Anweisung zur Ausführung des Blockes kann durch einen Methodenaufruf dieser neuen Klasse ersetzt werden. Dabei muß allerdings auch die gesamte Umgebung (Variablen und Parameter) mit übergeben werden.

(*Program*)	P	::=	$C_1 ... C_k\ E$
(*Class*)	C	::=	`class` $ClassId$
			`var` $Id_1 \ldots Id_k\ M_1 \ldots M_n$
			`end` $ClassId$
(*Method*)	M	::=	`method` $m_1 : Id_1 ... m_n : Id_n\ E$
(*Simple Expression*)	SE	::=	$ClassId$ `new` | `self` | Id |
			`nil` | *Integer*
(*Restricted Expression*)	RE	::=	SE | $RE\ m_1 : RE_1 ... m_n : RE_n$ |
			RE `instanceOf:` $ClassId$ |
			$RE_1\ comOp\ RE_2$| $RE_1\ arOp\ RE_2$
(*compareOperator*)	$comOp$	::=	$= \mid < \mid > \mid <= \mid >= \mid <>$
(*arithmeticOperator*)	$arOp$	::=	$+ \mid - \mid * \mid /$
(*Expression*)	E	::=	$RE \mid Id := RE \mid E_1; E_2 \mid$
			`if` RE `then` E_1 `else` E_2 `endif` |
			`while` RE `do` E `done` |
			`repeat` E `until` RE

Abbildung 2.1: Syntax der Programmiersprache SOL

$(m_1 : ...m_n :)$, seinen Parametern $(Id_1, ..., Id_n)$ und dem Methodenrumpf (E), einem Ausdruck. Ein Ausdruck kann entweder ein Instanziierungsausdruck, der die Erzeugung eines Objektes bewirkt, der vordefinierte Identifikator `self`, der undefinierte Wert `nil`, ein Methodenaufruf, ein `if-then-else`-Konstrukt, ein `repeat`-Schleifenkonstrukt oder ein `while`-Schleifenkonstrukt sein.

2.3 Laufzeitmodell

Ein SOL -Ausdruck wird in Abhängigkeit von der Klassenbibliothek und einem Zustand berechnet. Dieser Zustand besteht aus einem Heap (Objectmemory in Smalltalk-80 genannt) und einer Liste von *Activation Records* (Method Context in Smalltalk-80 genannt). Der Heap verwaltet die erzeugten Objekte, wobei ein Objekt eine Struktur bestehend aus Namen der zugehörigen Klasse und Werten für die Objekt-(Instanz-)variablen ist. Das *Activation Record* verwaltet die Bindungen der formalen Parameter an die aktuellen Argumente eines Methodenaufrufes und den Wert der `self`-Variablen sowie einen Zeiger auf das zuvor verwandte Activation Record[3], siehe Abbildung 2.2.

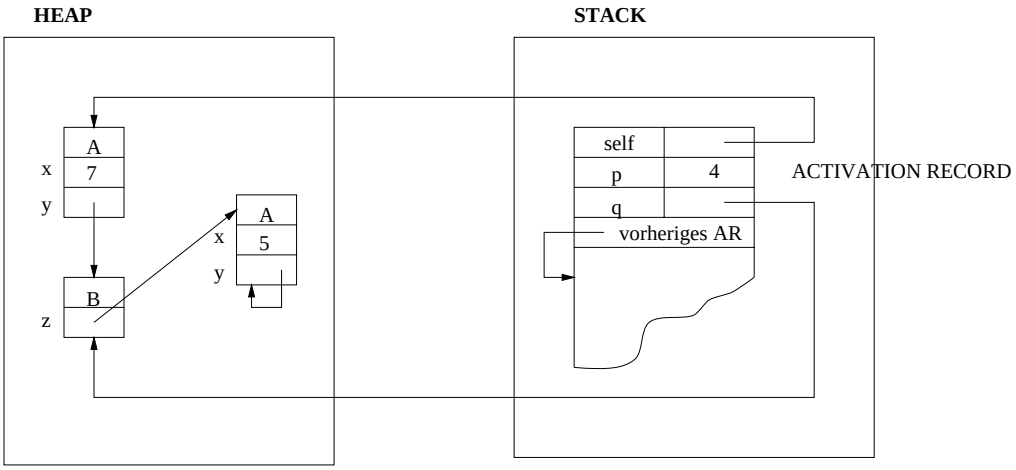


Abbildung 2.2: Laufzeitmodell

Im folgenden werden wir die Wirkungsweise der einzelnen Ausdrucksformen auf den Zustand beschreiben. Sei z der aktuelle Zustand:

integer|`nil` Hierbei wird der Zustand nicht verändert.

[3]Die Verwaltung von Methodenaufrufen erfolgt in der wohlbekannten Stack-Technik.

RE$_1$ *arOp* *RE*$_2$ Zunächst wird der Ausdruck *RE*$_1$ bzgl. des Zustandes z ausgewertet. Der resultierende Wert sei w_1 und der resultierende Zustand sei z_1. Dann wird der Ausdruck *RE*$_2$ bzgl. des Zustandes z_1 berechnet. Der daraus resultierende Wert sei w_2 und der resultierende Zustand sei z_2. Sofern *RE*$_1$ und *RE*$_2$ Integerzahlen darstellen, wird nun die entsprechende arithmetische Operation *arOp* $(+, -, *, /)$ auf die Werte w_1 und w_2 angewandt. Das Ergebnis dieser Operation und der Zustand z_2 werden als Gesamtresultat des Ausdruckes zurückgegeben.

RE$_1$ *comOp* *RE*$_2$ Entsprechend wie bei den arithmetischen Operationen.

`self` Es wird der Zeiger auf das momentane `self`-Objekt, abgelegt im Activation Record zurückgegeben. Der Zustand verändert sich nicht.

Id Der entsprechende Wert wird entweder im Activation Record oder bei den Instanzvariablen des Objektes auf das `self` zeigt, gefunden.

Id := *RE* Zunächst wird der Ausdruck *RE* ausgewertet. Sei w der resultierende Wert und z' der resultierende Zustand. Dann wird in z' der Instanzvariablen *Id* des Objektes, auf das `self` zeigt, der Wert w zugewiesen. Dieser derart geänderte Zustand sowie der Wert w bilden das Resultat des Gesamtausdruckes.

$E_1; E_2$ Zunächst wird der Ausdruck E_1 bzgl. des Zustandes z ausgewertet, sei z' der daraus resultierende Zustand. Dann wird der Ausdruck E_2 bzgl. des Zustandes z' berechnet. Der hieraus resultierende Wert sowie der resultierende Zustand sind das Ergebnis des Gesamtausdruckes.

RE `instanceOf:` *Id* Zunächst wird der Ausdruck *RE* bzgl. des Zustandes z berechnet, mit resultierendem Zustand z'. Nun wird überprüft, ob der Wert von *RE* Instanz der Klasse *Id* ist. Wenn ja, so wird der Wert `true` und z' zurückgegeben, ansonsten der Wert `false` und z'.

`if` *RE* `then` E_1 `else` E_2 `endif` Als erstes wird der Ausdruck *RE* berechnet, mit resultierendem Wert w_1 und Zustand z_1. Nun wird überprüft, ob w_1 ungleich dem Wert `false` ist. Wenn ja, dann wird der Ausdruck E_1 ausgewertet, ansonsten der Ausdruck E_2. Der daraus resultierende Wert und der daraus resultierende Zustand bilden das Ergebnis des Gesamtausdruckes.

`while` *RE* `do` *E* `done` Als erstes wird der Ausdruck *RE* berechnet, mit resultierendem Wert w_1 und Zustand z_1. Nun wird überprüft, ob w_1 ungleich dem Wert `false` ist. Wenn ja, dann wird der Ausdruck E ausgewertet; der daraus resultierende Wert und der daraus resultierende Zustand bilden nun die Basis für eine erneute Auswertung von *RE*, d.h. die Schleifeniteration wird fortgesetzt. Ansonsten sind w_1 und z_1 das Ergebnis des Gesamtausdruckes.

`repeat` *E* `until` *RE* Als erstes wird der Ausdruck E berechnet, mit resultierendem Wert w_1 und Zustand z_1. Dann wird der Ausdruck *RE* ausgewertet, mit resultierendem Wert w_2 und Zustand z_2. Nun wird überprüft, ob w_2 ungleich dem Wert `false` ist. Wenn ja, wird der Ausdruck E erneut ausgewertet, jetzt bzgl. des Zustandes z_2, d.h. die Schleifeniteration wird fortgesetzt. Ansonsten sind w_2 und z_2 das Ergebnis des Gesamtausdruckes.

$RE_0\ m_1 : RE_1\ \ldots\ m_n : RE_n$ Die Aktionen, die beim Auswerten eines solchen Ausdruckes erfolgen, lassen sich folgendermaßen skizzieren:

- Zunächst werden der Empfänger RE_0 und, falls vorhanden, die Argumente berechnet. Sei z' der daraus resultierende Zustand.
- Dann wird die Klassenzugehörigkeit des Empfängers bestimmt, sei K die entsprechende Klasse.
- Nun wird überprüft, ob in K eine Methode $m_1 : \ldots m_n :$ existiert. Falls nein, kommt es zum Programmabbruch mit der Fehlermeldung *does–not–understand*.
- Ansonsten wird ein neues Activation Record angelegt, in dem die Bindungen `self` $\rightarrow Wert(RE_0)$, $formalerParameter1 \rightarrow Wert(RE_1)$, ... , $formalerParameterN \rightarrow Wert(RE_n)$ und ein Verweis auf das alte Activation Record abgelegt sind.
- Nun kann der Ausdruck im Methodenrumpf von $m_1 : \ldots m_n :$ bzgl. des Zustands, bestehend aus dem Heap und dem neuen Activation Record, ausgewertet werden. Angenommen diese Berechnung terminiert mit dem Wert w und dem Zustand z''.

- Die Berechnung des Gesamtausdruckes endet damit, daß der Zustand z derart modifiziert wird, daß der Heap von z ersetzt wird durch den Heap von z''. Der Zustand z'' wird freigegeben. Der Wert des Gesamtausdruckes beträgt w.

2.4 Vorbereitende Umformungen

Programme der zuvor eingeführten syntaktischen Art eignen sich nicht sonderlich zur statischen Analyse (Datenflußanalyse). Daher werden SOL -Programme zunächst in "einfachere", genauer in einfacher zu analysierende, SOL –Programme transformiert. Hierbei werden alle Methodenaufrufe, die auf Empfänger- und Argumentpositionen in Methodenaufrufen vorkommen, eliminiert. Dies geschieht durch Einfügen zusätzlicher neuer Variablen. Diese im Programm nicht vorkommenden Variablen bekommen die zu eleminierenden Methodenaufrufe zugewiesen. Dann werden die Methodenaufrufe durch die entsprechenden Variablen ersetzt. Ähnlich werden Bedingungen, die Methodenaufrufe beinhalten, behandelt.

Beispiel:

Der Ausdruck `(o m:a) n:(p k:b)` wird ersetzt durch die Folge von Ausdrücken

```
t1 := o m:a;
t2 := p k:b;
t1 n:t2
```

wobei `t1` und `t2` neue Variablen sind. Entsprechend wird `if o n:a then b else c endif` ersetzt durch

```
t := o n:a;
if t then b else c endif
```

Programme, die diesen Anforderungen genügen, können wie in Abbildung 2.3 beschrieben werden.

Zur statischen Analyse wird meistens nicht die textuelle Darstellung von Programmen verwendet, sondern Programme werden als Graphen oder als eine Menge von Graphen repräsentiert. Wir werden drei geeignete Darstellungen in diesem Abschnitt einführen, das OO-Programm-Modell (Repräsentation eines Programms als Transistionssystem), das OO-Analyse-Modell (Aufsplitten der Methodenaufrufe,

(*Program*)	P	::=	$C_1 ... C_k\ E$
(*Class*)	C	::=	`class` $ClassId$
			`var` $Id_1 \ldots Id_k\ M_1 \ldots M_n$
			`end` $ClassId$
(*Method*)	M	::=	`method` $m_1 : Id_1 ... m_n : Id_n\ E$
(*Simple Expression*)	SE	::=	$ClassId$ `new` | `self` | Id |
			`nil` | *Integer*
(*Simple Method Call*)	SMC	::=	$SE_0\ m_1 : SE_1 ... m_n : SE_n$
(*Primitive Method Call*)	PMC	::=	$Id_1\ +\ Id_2 \mid Id_1\ -\ Id_2 \mid$
			$Id_1\ *\ Id_2 \mid Id_1\ /\ Id_2$
(*Simple Expression or Method Call*)	$SEMC$	::=	$SE \mid SMC \mid PMC$
(*Simple Condition*)	SC	::=	$SE \mid SE$ `instanceOf:` $ClassId \mid$
			$Id_1 = Id_2 \mid Id_1 < Id_2 \mid Id_1 > Id_2 \mid$
			$Id_1 <= Id_2 \mid Id_1 >= Id_2 \mid Id_1 <> Id_2$
(*Expression*)	E	::=	$SEMC \mid Id := SEMC \mid E_1; E_2 \mid$
			`if` SC `then` E_1 `else` E_2 `endif` |
			`while` SC `do` E `done` |
			`repeat` E `until` SC

Abbildung 2.3: Syntax der vereinfachten SOL -Programme

Herstellen des funktionalen Charakters eines Aufrufs, Einführen von Filtern) und den OO-Analyse-Graphen (Herstellen des interprozeduralen (inter-objekt) Datenflusses).

2.4.1 OO-Programm-Modelle

Um den Kontrollfluß und den Datenfluß in einem Programm darzustellen, werden Flußgraphen (flow graphs) verwendet. Dabei werden die Knoten des Graphen mit Ausdrücke dekoriert und die Kanten spiegeln den Kontrollfluß wieder. Jeder Prozedur oder Funktion wird ein solcher Flußgraph zugeordnet. Diese Technik ist schon seit längerem bekannt, siehe z.B. [MJ81], und kommt vor allen bei imperativen Sprachen zur Anwendung. Sie kann problemlos auf unsere objekt–orientierte Spra-

che SOL übertragen werden.

In der vorliegenden Arbeit werden Flußgraphen durch Transitionssysteme dargestellt [St93]. Hierbei werden nun die Kanten mit Ausdrücken der Sprache dekoriert und nicht die Knoten. Weiterhin modellieren die Kanten den Kontrollfluß einer Methode. Die Knoten repräsentieren nur die Programmpunkte.

Ein OO-Programm-Modell $T^{pm} =< M_1, ..., M_k, M_H >$ ist ein System von Methoden–Modellen, wobei diese die Methoden der Klassen des Programms darstellen. M_H ist ein künstlich hinzugefügtes Methoden-Modell, das den Ausdruck des Hauptprogramms darstellt.

Ein Methoden-Modell ist ein Transitionssystem $M = (N, E, s, e)$, wobei N eine Menge von Knoten (nodes), E eine Menge von Kanten (edges), s ein Knoten, der keinen Vorgänger besitzt, und e ein Knoten, der keinen Nachfolger besitzt, ist. Im weiteren werden wir im Hinblick auf den Sprachgebrauch bei Transitionssystemen Knoten auch als Zustände (states) und Kanten als Transitionen (transitions) bezeichnen.

O.B.d.A. habe der Endzustand jedes Methoden-Modells genau eine eingehende Transition. Dies kann ggf. durch Hinzufügen einer weiteren Transition erreicht werden. Somit sind Transitionen mit Ausnahme der Transition zum Endzustand eines Methoden–Modells dekoriert mit

- einfachen Ausdrücken (*SE*),
- einfachen oder primitiven Methoden (*SMC*, *PMC*),
- einfachen Bedingungen (*SC*),
- Zuweisungen, deren rechte Seite einfache Ausdrücke, einfache Methodenaufrufe oder primitive Methodenaufrufe sind (*SEMC*).

In der soeben beschriebenen Weise läßt sich ein SOL -Programm auf natürliche Weise als OO-Programm-Modell darstellen, wie am folgenden Beispielprogramm gezeigt wird.

Beispielprogramm:

```
class A
```

```
method m
  if 1<2
     then self
     else self m
  endif
end A

class B
var c d
method n
  c:=3;
  d:=4;
  while c<>d do
     c:=d
  done;
  self
method m
  A new n
end B

class Bsp
var a b
method run
  a := Bsp new;
  if a instanceOf: Bsp
     then a:= A new
     else a:= B new
   endif;
  b:=a m;
  b:=1
end Bsp

Bsp new run
```

Das zugehörige OO-Programm-Modell ist in Abbildung 2.4 dargestellt.

2.4.2 OO-Analyse-Modell

Ausgehend von einem OO-Programm-Modell erhalten wir das zugehörige OO-Analyse-Modell durch drei Transformationen. In einem ersten Schritt werden Methodenaufrufe aufgespalten (gesplittet). Dies geschieht derart, daß zunächst Empfänger und Argumente vor dem eigentlichen Aufruf separat ausgeführt werden und der Aufruf nur ihre Werte verwendet. In einem zweiten Schritt wird der funktionale

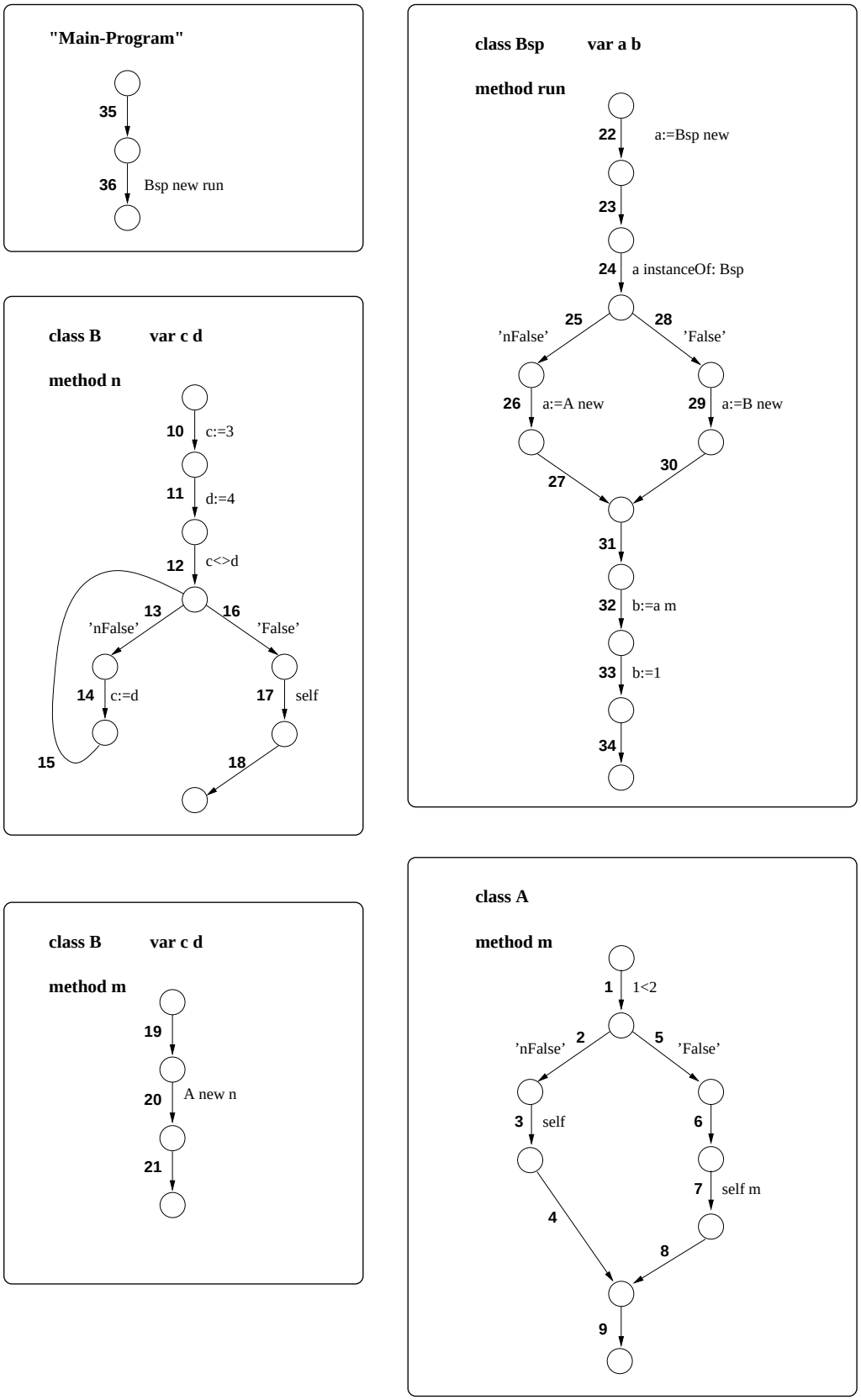


Abbildung 2.4: OO-Programm-Modell des Bespielprogramms

Charakter[4] der Methodenaufrufe explizit hergestellt. Das Ergebnis eines Methodenaufrufes wird in einer gesonderten Variablen abgelegt. Im letzten Schritt werden Bedingungen umgeformt, so daß sogenannte Filter bei der Analyse die geeigneten Zweige herausfiltern können. Es werden also nicht automatisch[5] beide Zweige eines `if-then-else`-Ausdruckes untersucht, sondern in Abhängigkeit der Bedingung evtl. nur ein Zweig. Diese Filtertechnik kommt auch später bei der Auswertung der Methodenaufrufe zum Selektieren des geigneten Methoden–Modells zur Anwendung.

Schritt 1: Aufsplitten der Methodenaufrufe

Jede Transition $e \equiv (s,t)$, die mit einem Ausdruck oder einer Zuweisung der Form

$$[Id :=]SE_0\ m_1 : SE_1 ... m_n : SE_n$$

dekoriert ist, wird durch Transitionen (s,r_0), (r_0,r_1),...,(r_{n-1},r_n), (r_n,t) ersetzt, die dekoriert sind mit *e-sendsTo* := SE_0, *e-param*$_1$:= SE_1,...,*e-param*$_n$:= SE_n und $[Id :=]$*e-sendsTo* m_1 : *e-param*$_1$...m_n : *e-param*$_n$, siehe Abbildung 2.5.

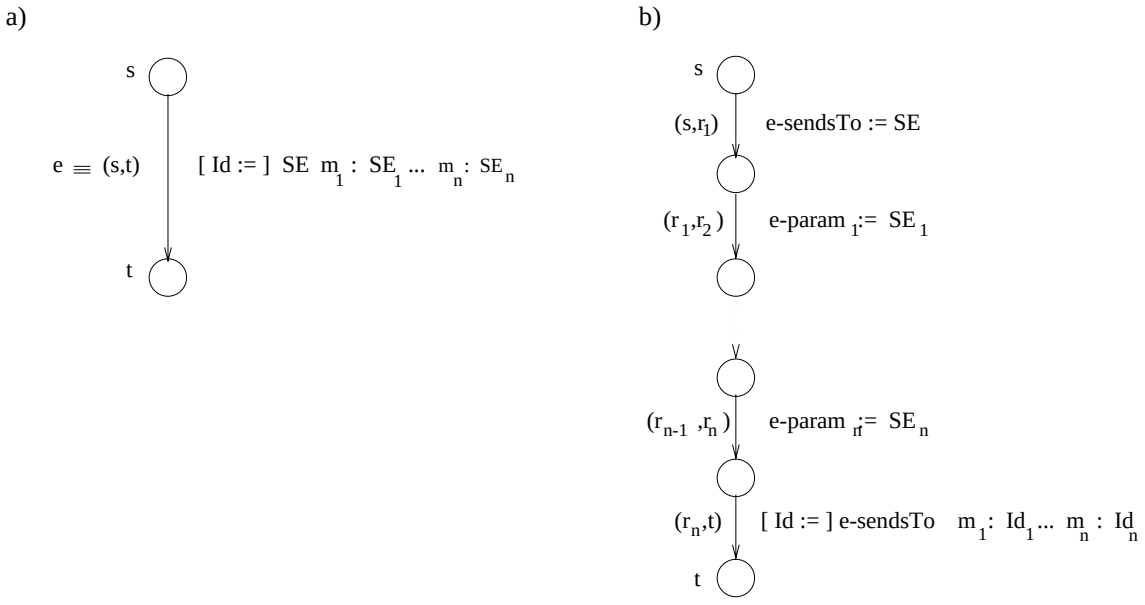


Abbildung 2.5: Aufsplitten der Methodenaufrufe

Die bereits im Programm definierten Variablen werden um *e-sendsTo* und *e-param*_1, ..., *e-param*$_n$ erweitert.

[4]Methodenaufrufe können, wie Funktionen, eine Wert zurückliefern.

[5]Dies hängt natürlich von der verwendeten Analyse ab, wie wir später sehen werden.

Schritt 2: Funktionalen Charakter herstellen

Erweitere die bereits im Programm definierten Variablen um die Variable *method-result*, die zur Speicherung des Wertes des zuletzt ausgewerteten Ausdrucks einer Methode verwendet wird. Um dies zu erreichen, wird nach folgendem Prinzip verfahren:

Sei die Menge der "Bearbeitungs" transitionen initialisiert durch die Transitionen, die in die Endzustände der Methode eingehen. Führe für alle Bearbeitungstransitionen aus:

1. Falls die Bearbeitungstransition mit einem einfachen Ausdruck oder Methodenaufruf *SEMC* dekoriert ist, dann ersetze die Dekorierung durch *method-result* := *SEMC*.

2. Falls die Bearbeitungstransition mit einer Zuweisung dekoriert ist, deren rechte Seite ein einfacher Ausdruck oder Methoden-Aufruf *SEMC* ist, dann dekoriere die vom Endzustand der Bearbeitungstransition ausgehenden Transitionen mit *method-result* := Id, wobei Id die linke Seite der Zuweisung ist.

3. Falls die Bearbeitungstransition nicht dekoriert ist, dann erweitere die Menge der Bearbeitungstransitionen um die in den Anfangszustand der Bearbeitungstransition eingehenden Transitionen.

4. Verringere die Menge der Bearbeitungstransitionen um die soeben bearbeitete Transition.

Algorithmisch läßt sich die Herstellung des funktionalen Charakters von Methoden beschreiben durch:

Algorithmus:

Sei $M = (N, E, s, e)$ ein Methoden-Modell.

$InT := InTransition(e)$
repeat
  for all $t \in InT$ do
    if $dekor(t) \equiv$ *SEMC* then
      $dekor(t) \equiv$ *method-result* $:=$ *SEMC*
    else if $dekor(t) \equiv Id :=$ *SEMC* then
      $dekor(OutTransition(dest(t))) \equiv$ *method-result* $:= Id$
    else $InT := InT \cup InTransition(source(t))$
    endif;
    $InT := InT \backslash \{t\}$
until $InT = \emptyset$

wobei $dekor(t)$ die Funktion ist, die zu der Transition t seine entsprechende Dekorierung liefert; $InTransition(z)$ die Funktion ist, die zu dem Zustand z die in ihm eingehenden Transitionen liefert; $OutTransition(z)$ die Funktion ist, die zu dem Zustand z die von ihm ausgehenden Transitionen liefert; $dest(t)$ die Funktion ist, die zu der Transition t seinen Endzustand liefert; $source(t)$ die Funktion ist, die zu der Transition t seinen Anfangszustand liefert.

Abbildung 2.6 illustriert die Herstellung des funktionalen Charakters.

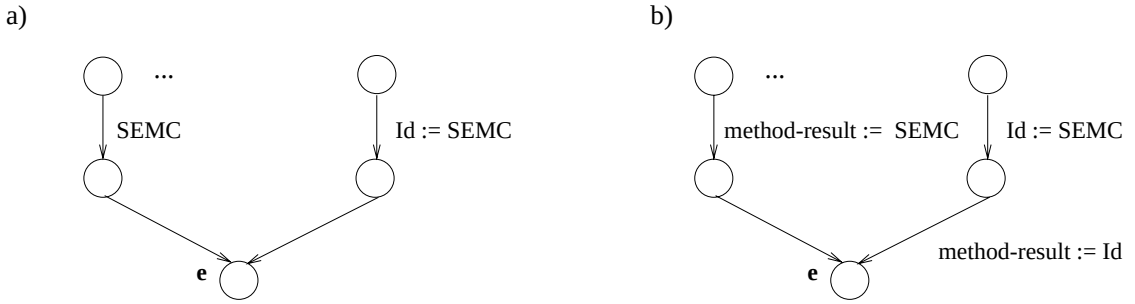


Abbildung 2.6: Funktionaler Charakter

Schritt 3: Einführung von Filtern

Die im Programm bereits vorkommenden Variablen werden um *e-cond* erweitert. Wir betrachten nun alle Transitionen, die mit Bedingungen dekoriert sind. Ähnlich wie in Schritt 1 wird die Bedingung durch Zuweisung an die Variable *e-cond* vor Ausführung des Verzweigungsausdruckes ausgewertet. D.h. jede Transition $e \equiv$

(s,t), die mit einer Bedingung der Form *SE* oder $Id_1 = Id_2$[6] dekoriert ist, wird nun mit *e-cond* := *SE* bzw. *e-cond* := $Id_1 = Id_2$ dekoriert. Jede Transition $e \equiv (s,t)$, die mit *SE* `instanceOf` :*Id* dekoriert ist, wird ersetzt (aufgesplittet) durch die Transitionen (s,r) und (r,t), die dekoriert werden mit *e-cond* := *SE* und *e-cond* := *e-cond* `instanceOf` :*Id*. Dann werden die vom Endzustand von e ausgehenden zwei Transitionen, die den '*nFalse*' bzw. '*False*' Zweig repräsentieren, mit *e-cond* `neqFalse` bzw. *e-cond* `eqFalse` dekoriert, siehe Abbildung 2.7.

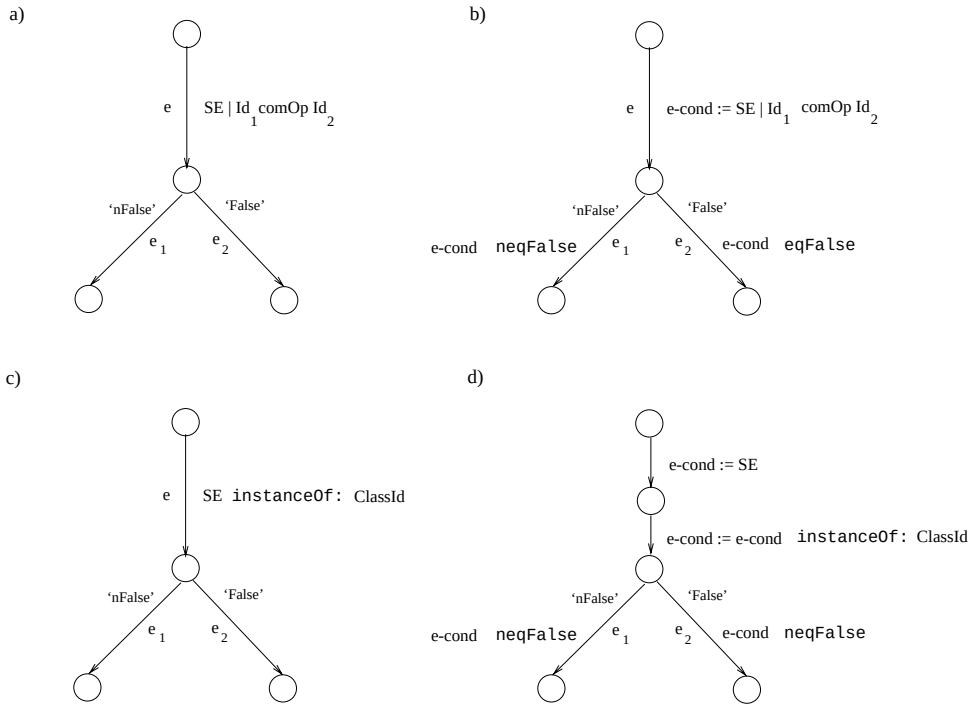


Abbildung 2.7: Einführen von Filtern

Das zum OO-Programm-Modell aus Abbildung 2.4 gehörige OO-Analyse-Modell ist in Abbildung 2.8 dargestellt.

Notation

Sei $T^{am} =< M_1, ..., M_k, M_H >$ das OO-Analyse-Modell eines Programmes Π.

- N^{am} bezeichne die Menge der Zustände von T^{am} und E^{am} bezeichne die Menge der Transitionen von T^{am}.

[6] Der Vergleichsoperator = sei hier stellvertretend auch für alle anderen Vergleichsoperatoren verwendet.

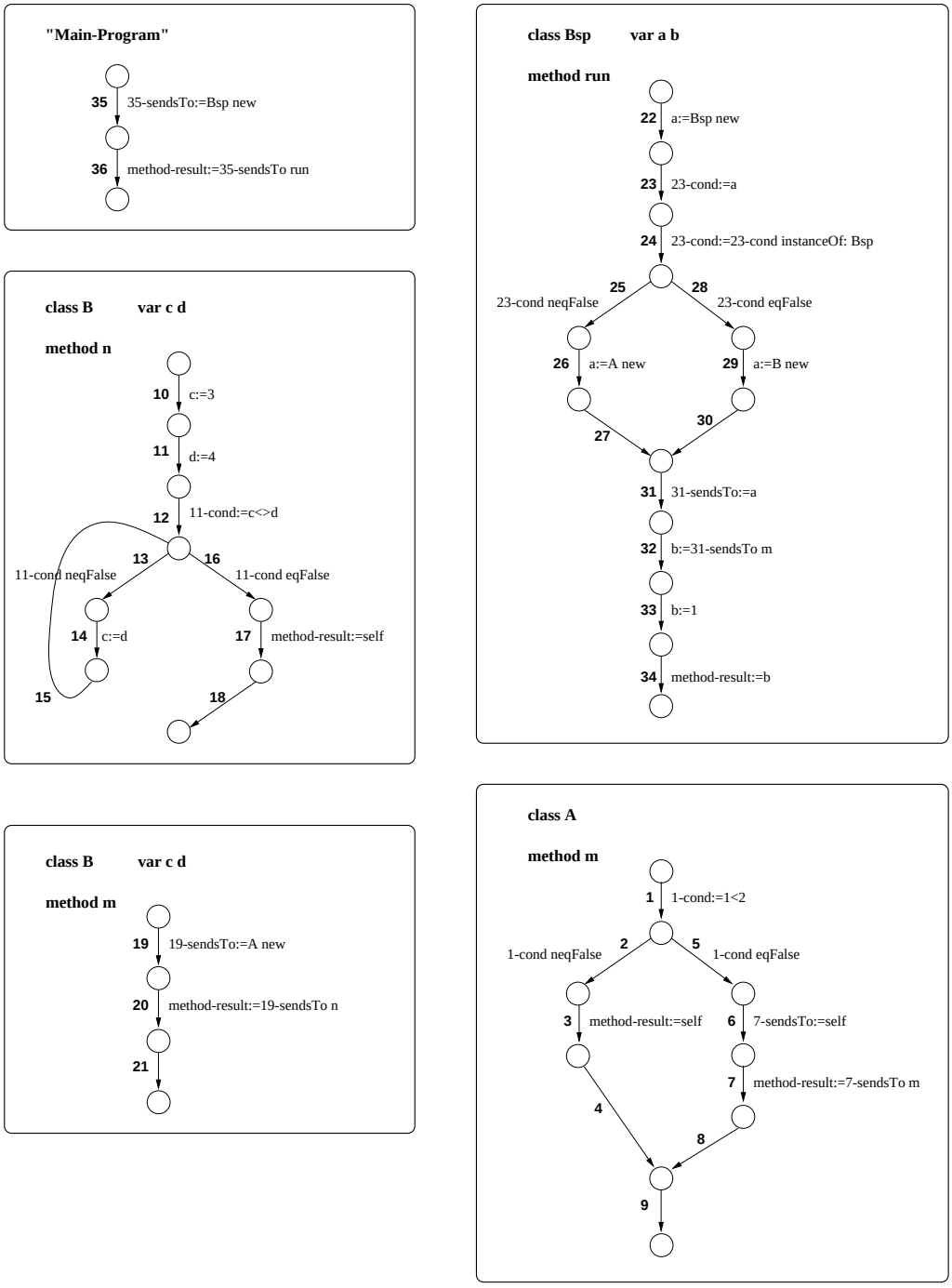


Abbildung 2.8: OO-Analyse-Modell des Bespielprogramms

- E^{am}_{pmc} bezeichne die Menge der Transitionen, die mit einem einfachen Methoden-Aufruf dekoriert sind; $E^{am}_{pmc} \subset E^{am}$.

- E^{am}_{assmc} bezeichne die Menge der Transitionen, die mit einer Zuweisung, in der ein einfacher Methoden-Aufruf vorkommt, dekoriert sind; $E^{am}_{assmc} \subset E^{am}$.

- E^{am}_{mc} bezeichne die Menge der Transitionen, in deren Dekorierung ein Methoden-Aufruf vorkommt; $E^{am}_{mc} =_{df} E^{am}_{pmc} \cup E^{am}_{assmc}$

- $\mathcal{C}_{Id}$ bezeichne die Menge der Klassenidentifikatoren der Klassen, die in Π definiert sind.

- $\mathcal{M}_{Id}$ bezeichne die Menge der Methodenidentifikatoren der Methoden, die in Π definiert sind.

- $\mathcal{V}_{Id}$ bezeichne die Menge der Variablenidentifikatoren, die in Π definiert sind, erweitert um die Menge
  $\{$*e-sendsTo*, *e-param*$_1$, ..., *e-param*$_n$, *method-result*, *e-cond*$\}$.

- $\mathcal{M}_{Mdl}$ bezeichne die Menge der Methoden-Modelle des OO-Analyse-Modells, d.h. $\mathcal{M}_{Mdl} = \{M_1, ..., M_k, M_H\}$.

- $Impl : \mathcal{M}_{Id} \rightarrow \mathcal{P}(\mathcal{C}_{Id})$ bildet einen Methodenidentifikator auf die Menge der Klassen ab, in denen eine solche Methode definiert ist.

- $Model : \mathcal{M}_{Id} \times \mathcal{C}_{Id} \rightarrow \mathcal{M}_{Mdl}$ bildet einen Methodenidentifikator und einen Klassenidentifikator auf das zugehörige Methodenmodell ab.

  $$Model(m, C) = M_i,$$

  M_i stellt das Methodenmodell der Methode m in der Klasse C dar.

- $Class : N^{am} \cup \mathcal{M}_{Mdl} \rightarrow \mathcal{C}_{Id}$ bildet Zustand und Methodenmodell auf die Klasse ab, zu der sie gehören;

  $$Class(y) = C,$$

  wobei C die Klasse darstellt, die y umfaßt.

- $mm : N^{am} \rightarrow \mathcal{M}_{Mdl}$ bildet einen Zustand auf dasjenige Methodenmodell ab, in dem der Zustand vorkommt;

$$mm(n) = M$$

wobei M das den Zustand n umfassende Methodenmodell ist.

- $name : \mathcal{M}_{Mdl} \rightarrow \mathcal{M}_{Id}$ liefert den Identifikator eines Methodenmodells;

$$name(M) = m,$$

wobei m der Methodenname des Methodenmodells ist.

- $Method : E^{am}_{mc} \rightarrow \mathcal{M}_{Id}$ bildet eine Transition aus E^{am}_{mc} auf dasjenige Methodenmodell ab, in dem die Transition vorkommt.

- $Sender : \mathcal{M}_{Mdl} \rightarrow \mathcal{P}(E^{am}_{mc})$ bildet ein Methodenmodell auf alle das Methodenmodell aufrufende Transitionen ab;

$$Sender(M) = \{e \in E^{am}_{mc} | Method(e) = name(M)\}.$$

- Sei im weiteren $M = (N, E, s, e)$ ein Methodenmodell von T^{am}, das zu einer Klasse C gehört.

- $start : \mathcal{M}_{Mdl} \rightarrow N$ und $end : \mathcal{M}_{Mdl} \rightarrow N$ liefern zu einem Methodenmodell den jeweiligen Start- bzw. Endzustand;

$$start(M) = s, \quad end(M) = e.$$

- $pred_M : N \rightarrow \mathcal{P}(N)$ liefert die Menge der direkten vorhergehenden Zustände von m;

$$pred_M(m) =_{df} \{n | (n, m) \in E\}.$$

- $succ_M : N \rightarrow \mathcal{P}(N)$ liefert die Menge der direkten nachfolgenden Zustände von n;

$$succ_M(n) =_{df} \{m | (n, m) \in E\}.$$

- Sei $e = (n, m) \in E$ eine Transition. Dann heißt n Anfangszustand von e und m Endzustand von e.
  $source : E \to N$ liefert den Anfangszustand und $dest : E \to N$ liefert den Endzustand einer Transition;

$$source(e) = n, \quad dest(e) = m.$$

- Ein endlicher Pfad in einem Graphen G ist eine Folge von Transitionen $(e_1, ..., e_q)$, so daß $dest(e_j) = source(e_{j+1})$ für alle $j \in \{1, ..., q-1\}$ gilt. Wir sagen auch: $(e_1, ..., e_q)$ ist ein Pfad von m nach n, wenn zusätzlich noch $source(e_1) = m$ und $dest(e_q) = n$ gilt.

- $P_G[m, n]$ bezeichne die Menge aller endlichen Pfade in einem Graphen G von m nach n. Der leere Pfad wird durch ϵ dargestellt, er enthält keine Transition.

- O.B.d.A. setzen wir voraus, daß alle Zustände $n \in N$ auf einem Pfad eines Methodenmodells von seinem Anfangszustand s nach seinem Endzustand e vorkommen.

2.4.3 OO-Analyse-Graph

Der Kontrollfluß zwischen Objekten, ausgelöst durch Methodenaufrufe, kann im OO-Analyse-Modell nicht ausgedrückt werden. Im OO-Analyse-Modell kann zwar jedes Methodenmodell separat analysiert werden (d.h. intra-objekt bzw. intraprozedurale Analyse), aber auf die Wechselwirkungen der Aufrufe kann nicht eingegangen werden. Daher werden wir nun das OO-Analyse-Modell derart verfeinern, daß der Kontrollfluß zwischen den Methodenmodellen integriert wird (d.h. es werden die Voraussetzungen für eine inter–objekt bzw. interprozedurale Analyse geschaffen).

Der OO-Analyse-Graph $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ ergibt sich aus dem OO-Analyse-Modell $T^{am} = (N^{am}, E^{am}, s^{am}, e^{am})$, indem alle vorkommenden Methodenaufrufe mit ihren entsprechenden Methodenmodellen verbunden werden. Dazu ersetzen wir im OO-Analyse-Modell T^{am} alle Transitionen $e = (r, t) \in E^{am}_{mc}$, die mit einem einfachen Methodenaufruf

$$[Id :=] e\text{-}sendsTo\ m_1 : e\text{-}param_1\ ...\ m_i : e\text{-}param_i$$

dekoriert sind durch die folgenden Transitionen, wobei $C_1, ..., C_k$ diejenigen Klassen sind, in denen eine Methode $m_1 : ...m_i :$ definiert ist:

$e_{fil}^{C_1} \equiv (r, r_1)$, dekoriert mit *e-sendsTo* $C_1.m_1 : ...m_i :$,

$\vdots$,

$e_{fil}^{C_k} \equiv (r, r_k)$, dekoriert mit *e-sendsTo* $C_k.m_1 : ...m_i :$,

$e_{SthOutOf} \equiv (r, r_{k+1})$, dekoriert mit *e-sendsTo SthOutOf*$\{C_1, ..., C_k\}$,

und

$e_{pb}^{C_1} \equiv (r_1, start(Model(m_1 : ...m_i :, C_1)))$ dekoriert mit

$$(f_1^{C_1}, ..., f_i^{C_1}) := (e\text{-}param_1, ..., e\text{-}param_i),$$

$\vdots$

$e_{pb}^{C_k} \equiv (r_k, start(Model(m_1 : ...m_i :, C_k)))$ dekoriert mit

$$(f_1^{C_k}, ..., f_i^{C_k}) := (e\text{-}param_1, ..., e\text{-}param_i),$$

und

$(end(Model(m_1 : ...m_i :, C_1)), t)$ ohne Dekorierung im Falle $e \in E_{pmc}^{am}$ und mit Dekorierung $Id :=$ *method-result* im Falle $e \in E_{assmc}^{am}$,

$\vdots$

$(end(Model(m_1 : ...m_i :, C_k)), t)$ ohne Dekorierung im Falle $e \in E_{pmc}^{am}$ und mit Dekorierung $Id :=$ *method-result* im Falle $e \in E_{assmc}^{am}$

sowie

einer Transition (r_{k+1}, t) ohne Dekorierung.

Abbildung 2.9 und 2.10 illustrieren graphisch den soeben beschriebenen Vorgang. Im einzelnen bedeutet dies, daß anstelle der ursprünglichen Transition $e \in E_{mc}^{am}$ für alle entsprechenden Methodenimplementierungen ($Impl(Method(e)) = \{C_1, ..., C_k\}$) eine neue "Filter"-Transition e_{fil}^{C} eingefügt wird. Diese Transition dient später dazu herauszufinden, ob der Empfänger (*e-sendsTo*) Instanz der Klasse C ist. Wenn ja, so wird die Methode der Klasse C ausgeführt, wenn nein, so wird der Informationsfluß für diese Methode geblockt (d.h. die Methode wird nicht ausgeführt). Der Endzustand der Filter-Transition wird mit dem Anfangszustand des geeigneten Methodenmodells in C verbunden durch eine weitere Transition, die die Bindung der formalen Parameter $(f_1^C, ..., f_i^C)$ mit den aktuellen Parameter (Argumenten) erledigt. Weiterhin wird der Endzustand des Methodenmodells verbunden mit dem Endzustand der ursprünglichen Transition e.

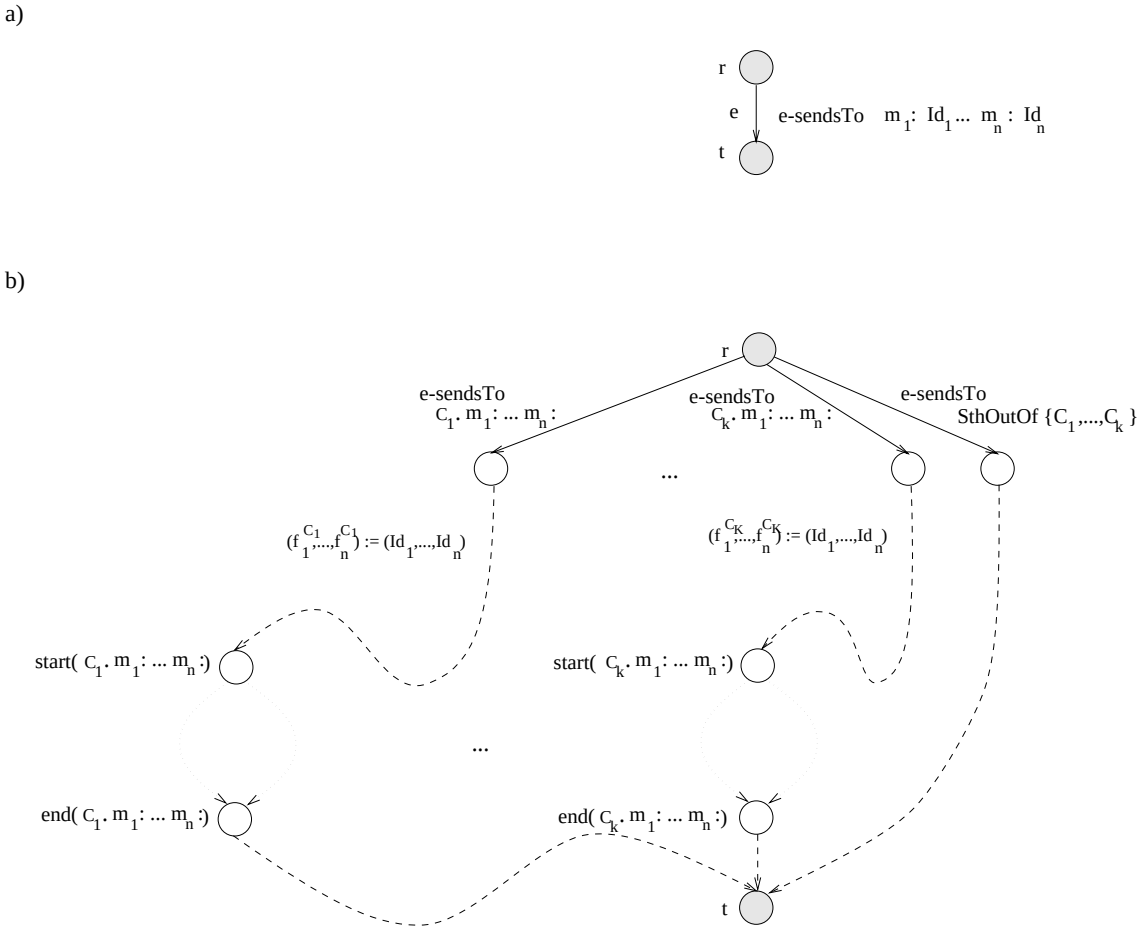


Abbildung 2.9: Inter-Objekt-Fluß für einfache Methodenaufrufe

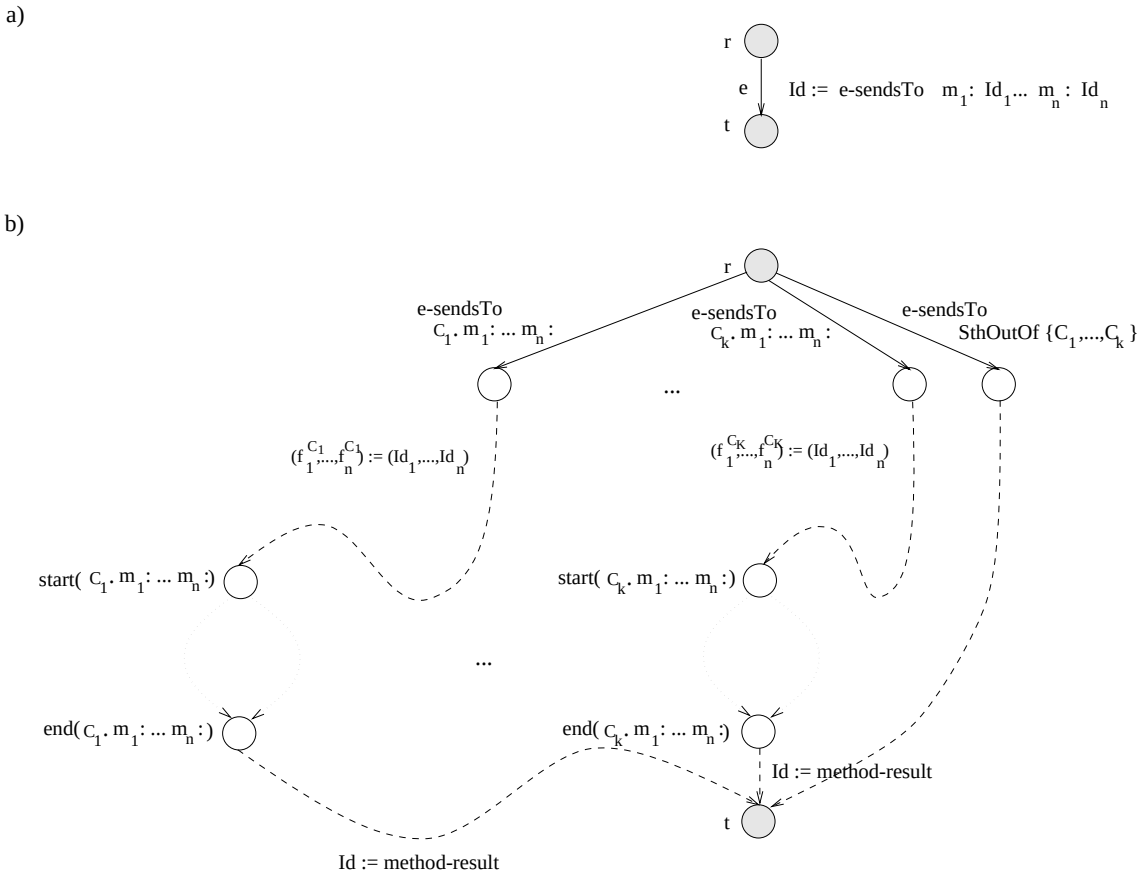


Abbildung 2.10: Inter-Objekt-Fluß für Zuweisungen mit einfachen Methodenaufrufen

Im Abbildung 2.10 wird der Fall illustriert, daß die ursprüngliche Transition e mit einer Zuweisung dekoriert ist, in der ein Methoden-Aufruf vorkommt. Hierbei wird zusätzlich zum Vorhergehenden noch die "Rückkehr"-Transition, ausgehend vom Endzustand des Methodenmodells hin zum Endzustand der ursprünglichen Transition, dekoriert mit der Zuweisung $Id := \textit{method-result}$. Damit wird die Verwertung der Wertrückgabe einer Methode realisiert.

Bezeichnungen:

- Aufruf-Transitionen sind diejenigen Transitionen, die bei der Konstruktion des OO-Analyse-Graphen eingefügt worden sind und die vom Anfangszustand der ursprünglichen Transition ausgehen ($e_{fil}^{C_1}, ..., e_{fil}^{C_k}, e_{SthOutOf}$ sind solche Aufruf-Transitionen). E_{call}^{ag} bezeichne die Menge aller Aufruf-Transitionen von T^{ag}.

- Rückkehr-Transitionen sind diejenigen Transitionen, die bei der Konstruktion des OO-Analyse-Graphen eingefügt worden sind und deren Endzustand dem Endzustand der ursprünglichen Transition entspricht. E_{ret}^{ag} bezeichne die Menge aller Rückkehr-Transitionen von T^{ag}.

- Die Indizes pb, fil, $SthOutOf$ bzw. ass bei den Transitionen weisen daraufhin, daß es sich um Transitionen handelt, die die Parameterbindungen, den Filter, den Ausnahmefall bzw. die Wertrückgabe eines Methodenaufrufs darstellen.

Gültige und vollständige Pfade

Endliche Pfade sind zuvor definiert worden. Allerdings reichen diese nicht aus, um das korrekte Aufruf- und Rückkehrverhalten von Methoden darzustellen. Es gibt endliche Pfade, die keine korrekten Programmausführungen beschreiben. Daher werden wir spezielle endliche Pfade (gültige Pfade) charakterisieren, die das Aufruf- und Rückkehrverhalten von Methoden in geeigneter Weise modellieren. Später benötigen wir noch gültige Pfade mit der Eigenschaft, daß auf ihnen genau soviele Aufruf-Transitionen wie auch Rückkehr-Transitionen vorkommen. Diese Pfade werden vollständige Pfade genannt. Die formalen Definitionen gehen zurück auf die Definitionen von interprozeduralen gültigen und vollständigen Pfaden aus [SP81].

Definition 2.4.1

1. Sei $p \in P_{T^{ag}}[m,n]$. Dann ist p ein gültiger Pfad (oder auch oo-Pfad genannt) genau dann, wenn das Tupel $(e_1, ..., e_k)$, das durch Elimination aller Transitionen aus $E^{ag} \backslash (E^{ag}_{call} \cup E^{ag}_{ret})$ wohlgeformt im folgenden Sinne ist:

   (a) Wenn in $(e_1, ..., e_k)$ keine Rückkehr-Transition vorkommt (d.h. $\forall e_i \in \{e_1, ..., e_k\} : \; e_i \notin E^{ag}_{ret}$), dann ist $(e_1, ..., e_k)$ wohlgeformt.

   (b) Ansonsten, sei j der kleinste Index aus $\{1, ..., k\}$, so daß e_j eine Return–Transition ist (d.h. $\forall i < j : \; e_i \notin E^{ag}_{ret}$). Dann ist $(e_1, ..., e_k)$ wohlgeformt genau dann, wenn $j > 1$ und $e_{j-1} \in E^{ag}_{call}$ die Aufruf–Transition ist, die der Return–Transition e_j entspricht und wenn das nach Eliminierung von e_{j-1} und e_j verbleibende Tupel $(e_1, ..., e_{j-2}, e_{j+1}, ..., e_k)$ wohlgeformt ist.

2. $OP[m,n]$ bezeichne die Menge aller oo-Pfade von m nach n.

3. Ein oo-Pfad $p = (e_1, ..., e_k) \in OP[source(e_1), dest(e_k)]$ ist ein vollständiger Pfad, wenn genau soviele Aufruf- wie Rückkehr-Transitionen in p vorkommen, d.h.

$$| \{i \,|\, e_i \in E^{ag}_{call}\} | = | \{i \,|\, e_i \in E^{ag}_{ret}\} |$$

4. $COP[m,n]$ bezeichne die Menge aller vollständigen (complete) oo-Pfade von m nach n.

Kapitel 3

Datenflußanalyse-Rahmen

Der hier vorgestellte Datenflußanalyse-Rahmen geht zurück auf den Rahmen der Abstrakten Interpretation [CC77, CC79, Ma93], einer mächtigen und mathematisch formulierten Theorie als Basis für statische Programmanalyse. Hierbei wird die "volle (dynamische)" Semantik ersetzt durch eine einfachere (abstraktere) Version, die speziell auf das vorliegende Problem zugeschnitten ist. Kennzeichnend für den hier definierten Datenflußanalyseansatz, eine Adaption des interprozeduralen Ansatz von Knoop und Steffen [Kn93, KS92] für imperative Sprachen, ist die Einteilung in

- Spezifikation des Datenflußanalyseproblems,
- Berechnung der Datenflußinformationen mittels eines generischen Algorithmuses
- und dem Nachweis gewisser Eigenschaften, um Korrektheit oder Optimalität (Koinzidenz) der Analyse zu gewährleisten.

Zur Spezifikation des gewünschten Datenflußanalyseproblems genügt es, einen vollständigen Verband, der die Datenflußinformationen repräsentiert, und die (lokale) abstrakte Semantik von "einfachen" Ausdrücken[1] zu definieren.

Die Berechnung der Datenflußinformationen geht zurück auf den "funktionalen" Ansatz von Sharir und Pnueli [SP81], bei dem Prozeduren als Einheiten (Blöcke) betrachtet werden, zu denen Ein-/Ausgabetransformationen berechnet werden. In

[1] Dies sind Ausdrücke, in denen keine Methodenaufrufe vorkommen.

einem zweiten Schritt wird dann das gesamte Programm analysiert, wobei die Prozeduraufrufe mittels ihrer bereits berechneten Semantik (Ein-/Ausgabetransformation) ausgewertet werden. Sowohl die Analyse der Prozeduraufrufe als auch des gesamten Programms erfolgt als iterative Fixpunktberechnung.

Mit Hilfe der lokalen abstrakten Semantik kann eine, allerdings nur theoretisch verwendbare, Lösung des Datenflußanalyseproblems definiert werden. Diese "operationelle" Lösung bildet die sogenannte Referenzlösung. Dahingegen kann die Fixpunktlösung des Datenflußanalyseproblems algorithmisch umgesetzt, also berechnet, werden. Gewisse Bedingungen, wie Nachweis der absteigenden Kettenbedingung des zugrundeliegenden Verbandes und der Nachweis der Monotonie bzw. Distributivität der lokalen abstrakten Semantik, führen zur Korrektheit bzw. Koinzidenz der Fixpunktlösung bzgl. der Referenzlösung.

Erwähnt werden soll noch, daß in diesem Kapitel kein konkretes Datenflußanalyseproblem definiert wird, somit auch kein konkreter Verband, sondern "nur" der Rahmen für eine Datenflußanalyse von objekt–orientierten Programmen. Dies bedeutet, daß dieser Rahmen über die in dieser Arbeit vorgestellten Typanalysen hinaus auch für andere Datenflußanalyseprobleme, wie z.B. *code motion, dead code elimination, reaching definitions* [Zi82, WM95, KG96a], verwendet werden kann, ohne jeweils vollständig neue Lösungen und Implementierungen mit dem Nachweis ihrer Korrektheit entwickeln zu müssen.

3.1 Verbände

Der der Verbandstheorie kundige Leser kann diesen Abschnitt beruhigt überschlagen. Die folgenden Definitionen gehen zurück auf [Sc86].

Definition 3.1.1
Eine Relation $\sqsubseteq: D \times D \to \mathbb{B}$ ist eine partielle Ordnung (*partial odering*) über dem Bereich (*domain*) D genau dann, wenn folgende Axiome für $\sqsubseteq$ gelten:

1. Reflexivität: für alle $a \in D$ gilt: $a \sqsubseteq a$.
2. Antisymmetrie: für alle $a, b \in D$ gilt: $a \sqsubseteq b$ und $b \sqsubseteq a$ impliziert $a = b$.
3. Transitivität: für alle $a, b, c \in D$ gilt: $a \sqsubseteq b$ und $b \sqsubseteq c$ impliziert $a \sqsubseteq c$.

Die in der reellen Analysis bekannten Begriffe "kleinstes" und "größtes" Element sowie "obere" und "untere" Schranke können in diesem Zusammenhang ebenfalls verwendet werden.

Definition 3.1.2
Sei D eine Menge mit partieller Ordnung $\sqsubseteq$ über D und $D' \subseteq D$.

- Ein Element $a \in D$ heißt kleinstes (bottom-) Element von D, falls gilt: $a \sqsubseteq b$ für alle $b \in D$.
- Ein Element $a \in D$ heißt größtes (top-) Element von D, falls gilt: $b \sqsubseteq a$ für alle $b \in D$.
- Ein Element $s \in D$ heißt obere Schranke von D', falls gilt: $a \sqsubseteq s$ für alle $a \in D'$.
- Ein Element $s \in D$ heißt untere Schranke von D', falls gilt: $s \sqsubseteq a$ für alle $a \in D'$.

Als nächstes führen wir Operationen auf D ein: eine Vereinigungsoperation $\sqcup$ (auch *join* genannt) und eine Schnittoperation $\sqcap$ (auch *meet* genannt).

Definition 3.1.3
Sei D eine Menge mit partieller Ordnung $\sqsubseteq$ über D und für alle $a, b \in D$ bezeichne der Ausdruck $a \sqcup b$ bzw. $a \sqcap b$ dasjenige Element in D (falls ein solches existiert), so daß gilt:

1. $a \sqsubseteq a \sqcup b$ und $b \sqsubseteq a \sqcup b$.
2. Für alle $d \in D$ gilt: $a \sqsubseteq d$ und $b \sqsubseteq d$ impliziert $a \sqcup b \sqsubseteq d$.

bzw.

1. $a \sqcap b \sqsubseteq a$ und $a \sqcap b \sqsubseteq b$.
2. Für alle $d \in D$ gilt: $d \sqsubseteq a$ und $d \sqsubseteq b$ impliziert $d \sqsubseteq a \sqcap b$.

Nun können wir einen Verband definieren.

Definition 3.1.4
Ein Verband $(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq)$ besteht aus:

- einer Menge $\mathcal{L}$ mit partieller Ordnung $\sqsubseteq$ und
- auf $\mathcal{L}$ sind join- und meet-Operationen definiert, d.h. $\forall a, b \in \mathcal{L}$ gilt: $a \sqcup b \in \mathcal{L}$ und $a \sqcup b \in \mathcal{L}$.

Die join- und meet-Operationen lassen sich auf Mengen von Argumenten erweitern.

Definition 3.1.5
Sei D eine Menge mit partieller Ordnung $\sqsubseteq$ über D und $X \subseteq D$. Dann bezeichne $\bigsqcup X$ (*least upper bound, lub*) bzw. $\bigsqcap X$ (*greatest lower bound, glb*) dasjenige Element (falls ein solches existiert), für das gilt:

1. Für alle $x \in X$ gilt: $x \sqsubseteq \bigsqcup X$.
2. Für alle $d \in D$ gilt: Wenn für alle $x \in X$: $x \sqsubseteq d$, dann $\bigsqcup X \sqsubseteq d$.

bzw.

1. Für alle $x \in X$ gilt: $\bigsqcap X \sqsubseteq x$.
2. Für alle $d \in D$ gilt: Wenn für alle $x \in X$: $d \sqsubseteq x$, dann $d \sqsubseteq \bigsqcap X$.

Damit haben wir alle Vorbereitungen für die Definition eines vollständigen Verbandes abgeschlossen.

Definition 3.1.6
Ein vollständiger Verband $(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ besteht aus:

- einem Verband $(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq)$,
- für alle Teilmengen $X \subseteq \mathcal{L}$ existieren $\bigsqcup X$ und $\bigsqcap X$ und
- $\bot = \bigsqcap \mathcal{L}$ bezeichnet das kleinste (bottom-) Element und $\top = \bigsqcup \mathcal{L}$ bezeichnet das größte (top-) Element des Verbandes.

Der Bereich der Datenflußinformationen, die wir später verwenden, bildet einen vollständigen Verband (complete lattice), dessen Elemente die gewünschten Datenflußinformationen ausdrücken und der die absteigende Kettenbedingung erfüllt. Daher führen wir noch einen Begriff ein.

Definition 3.1.7
Ein (vollständiger) Verband $(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ erfüllt die absteigende Kettenbedingung (descending chain condition) genau dann, wenn für jede Teilmenge von $L \subseteq \mathcal{L}$ und jede Folge von Elementen aus L mit

$$c_1 \sqsupseteq c_2 \sqsupseteq c_3 \sqsupseteq c_4 \sqsupseteq ...$$

gilt: es gibt einen Index k, so daß

$$\forall j \geq k : \; c_j = c_k.$$

3.2 Lokale Abstrakte Semantik

Sei Π ein SOL -Programm, $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph, dann definieren wir die lokale abstrakte Semantik durch ein Funktional

$$[\![\;\;]\!]^{loc} : E^{ag} \to (\mathcal{L} \to \mathcal{L}),$$

das jeder Transition des OO-Analyse-Graphen eine abstrakte Bedeutung zuordnet, ausgedrückt in Transformationen des vollständigen Verbandes. Wie dies im einzelnen erfolgen kann, werden wir in Kapitel 4 sehen.

Die globale abstrakte Semantik eines Programmes kann auf zweierlei Weise beschrieben werden: durch den Meet-Over-all-Path-Ansatz (*OMOP*) und durch den Maximal-Fixed-Point-Ansatz (*OMFP*).

3.3 Der *OMOP*-Ansatz

Die Definition des lokalen abstrakten Semantik–Funktionals kann ohne weiteres auf endliche Pfade ausgeweitet werden. Für jeden Pfad $p \in OP[m, n]$ definieren wir

$$[\![\;\;]\!]^* : \; (E^{ag})^* \; \to (\mathcal{L} \to \mathcal{L}) \; mit$$

$$[\![p]\!]^* = \begin{cases} \mathsf{Id}_{\mathcal{L}} & \text{falls } p = \epsilon \\ [\![(e_2, ..., e_k)]\!]^* \circ [\![e_1]\!]^{loc} & \text{sonst (mit } p = (e_1, ..., e_k)) \end{cases}$$

Beim *OMOP*-Ansatz werden alle möglichen Programmausführungen separat durchgespielt. Die an jedem Programmpunkt errechneten Informationen werden abschließend durch die meet–Operation zusammengefaßt.

Definition 3.3.1
Sei Π ein SOL -Programm und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann wird die *OMOP*-Lösung bezüglich eines vollständigen Verbandes $\mathcal{L}$, einer Startinformation $c_0 \in \mathcal{L}$ und einer lokalen Semantik $[\![\ ]\!]^{loc}$ definiert durch:

$$\forall\, c_0 \in \mathcal{L}\ \forall\, n \in N^{ag} :\ OMOP_{([\![\ ]\!]^{loc}, c_0)}(n) =_{df} \bigsqcap \{\, [\![p]\!]^*(c_0) \mid p \in \mathbf{OP}[s^{ag}, n] \,\}$$

Leider eignet sich diese Definition nicht für eine reale, effektive Implementierung, da im Falle von rekursiven Methodenaufrufen es unendlich viele Pfade geben würde.

3.4 Der *OMFP*-Ansatz

Bei der Anwendung des Maximal-Fixed-Point-Ansatzes zur interprozeduralen Datenflußanalyse werden die Bedeutungen von Prozeduren zunächst in einem Präprozeß berechnet. Somit können die Bedeutungen von Prozeduraufrufen mit den bereits berechneten Prozedurbedeutungen ausgedrückt werden. Diese Strategie übertragen wir für objekt–orientierte Programme.

Um die Bedeutung von Methodenmodellen zu beschreiben, führen wir zuerst ein weiteres Semantik–Funktional $[\![\ ]\!]$ ein. $[\![n]\!]$ bewirkt, daß die Datenflußinformation, die zu Anfang des n umfassenden Methodenmodells gültig ist, transformiert wird in eine Datenflußinformation, die gültig bei n ist. Insgesamt erhalten wir damit, daß die Bedeutung einer Methode m durch $[\![end(m)]\!]$ ausgedrückt werden kann. Da in der SOL -Sprache Methodenaufrufe immer Werte zurückliefern, erweitern wir die interprozeduralen Definitionen aus [SP81, KS92] derart, daß Zuweisungen, auf deren rechten Seiten Methodenaufrufe vorkommen, korrekt behandelt werden.

Gleichungssystem 3.4.1
Sei Π ein SOL -Programm und $T^{am} = (N^{am}, E^{am}, s^{am}, e^{am})$ das zugehörige OO-Ana-

lyse-Modell. Dann sind die Funktionale $[\![\ ]\!] : N^{am} \to (\mathcal{L} \to \mathcal{L})$ and $[\ ] : E^{am} \to (\mathcal{L} \to \mathcal{L})$ definiert als die größte Lösung des Gleichungssystems:

$$[\![\ n\ ]\!] = \begin{cases} Id_{\mathcal{L}} & \text{falls } n \in start(\mathcal{M}_{Mdl}) \\ \bigsqcap\{[\ (m,n)\ ] \circ [\![\ m\ ]\!] \mid m \in pred_{mm(n)}(n)\} & \text{sonst} \end{cases}$$

und

$$[\ e\ ] = \begin{cases} [\ e_{SthOutOf}\ ]^{loc} \sqcap \\ \bigsqcap\{ [\ e^C_{ass}\ ]^{loc} \circ [\![\ end(Model(Method(e), C))\ ]\!] \circ [\ e^C_{pb}\ ]^{loc} \circ [\ e^C_{fil}\ ]^{loc} \\ \qquad\qquad \mid\ C \in Impl(Method(e))\ \} \\ \qquad\qquad \text{falls } e \in E^{am}_{assmc} \\ [\ e_{SthOutOf}\ ]^{loc} \sqcap \\ \bigsqcap\{ [\![\ end(Model(Method(e), C))\ ]\!] \circ [\ e^C_{pb}\ ]^{loc} \circ [\ e^C_{fil}\ ]^{loc} \\ \qquad\qquad \mid\ C \in Impl(Method(e))\ \} \\ \qquad\qquad \text{falls } e \in E^{am}_{pmc} \\ [\ e\ ]^{loc} \qquad\qquad \text{sonst} \end{cases}$$

wobei $\mathsf{Id}_{\mathcal{L}}$ die Identität auf $\mathcal{L}$ bezeichnet und $\bigsqcap$ die komponentenweise Schnittoperation (meet) auf dem Funktionenverband $[\mathcal{L} \to \mathcal{L}]$ ist.

In Worten ausgedrückt, die Semantik einer Aufruf-Transition $e \in E^{am}_{mc}$ wird durch den Schnitt über die Bedeutungen aller geeigneten Methodendefinitionen, die in den in Π vorhandenen Klassen vorkommen, beschrieben. Genauer betrachtet werden folgende Schritte ausgeführt:

1. Filtern des Aufrufes:
   $[\ e^C_{fil}\ ]^{loc}$ propagiert Informationen nur dann, wenn die Methode der Klasse C auch ausgeführt werden könnte.

2. Binden der aktuellen Argumente an die formalen Parameter:
   $[\ e^C_{pb}\ ]^{loc}$ führt diese Bindung aus.

3. Berechnung des Aufrufes:
   $[\![\ end(...)\ ]\!]$ berechnet die Bedeutung des Methodenrumpfes.

4. Falls $e \in E^{am}_{assmc}$, dann Wertrückgabe:
   $[\![e^{C}_{ass}]\!]^{loc}$ weist der linken Seite der dargestellten Zuweisung das Resultat des Methodenaufrufs zu.

5. Behandlung des Ausnahmefalles:
   $[\![e_{SthOutOf}]\!]^{loc}$ überprüft, ob ein Methodenaufruf korrekt ist.

Ein weiteres Gleichungssystem berechnet nun die gewünschten Datenflußinformationen des gesamten Programms.

Gleichungssystem 3.4.2

$$dfi(n) \quad = \quad \begin{cases} c_0 & \text{falls } n = start(M_H) \\ \bigsqcap \{ [\![e^{Class(n)}_{pb}]\!]^{loc} \circ [\![e^{Class(n)}_{fil}]\!]^{loc}(dfi(source(e))) \mid e \in Sender(mm(n)) \} & \\ & \text{falls } n \in start(\mathcal{M}_{Mdl}) \backslash \{start(M_H)\} \\ \bigsqcap \{ [\![(m,n)]\!](dfi(m)) \mid m \in pred_{mm(n)}(n) \} & \text{sonst} \end{cases}$$

Somit können wir nun die *OMFP*-Lösung definieren.

Definition 3.4.3
Sei Π ein SOL -Programm, $T^{am} = (N^{am}, E^{am}, s^{am}, e^{am})$ das zugehörige OO-Analyse-Modell und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann wird die *OMFP*-Lösung bezüglich eines vollständigen Verbandes $\mathcal{L}$, einer Startinformation $c_0 \in \mathcal{L}$ und einer lokalen Semantik $[\![\ ]\!]^{loc}$ definiert durch[2]:

$$\forall\, c_0 \in \mathcal{L}\ \forall\, n \in N^{ag} :\ OMFP_{([\![\ ]\!]^{loc}, c_0)}(n) =_{df} dfi_{c_0}(n)$$

3.5 Informelle Beschreibung der *OMFP*-Lösung

Die *OMFP*-Lösung wird durch zwei Gleichungssysteme 3.4.1 und 3.4.2 beschrieben. Zunächst wird die abstrakte Semantik von Methoden berechnet, siehe 3.4.1. Danach werden die hierbei erzielten Resultate für die Analyse des gesamten Programmes, d.h. der Berechnung der Datenflußinformationen, verwendet. Ausgehend von den Definitionen 3.4.1 und 3.4.2 können wir die folgenden informellen Beschreibungen zur

[2] dfi_{c_0} bezeichnet die größte Lösung des Gleichungssystems 3.4.2 bzgl. der Startinformation c_0.

Berechnung der *OMFP*-Lösung entwickeln. Hierbei verwenden wir (a) eine globale Transformation $gtr : N^{am} \to \mathcal{L} \to \mathcal{L}$, die das $[\![\ ]\!]$-Funktional darstellt; (b) eine lokale Transformation $ltr : E^{am} \to \mathcal{L} \to \mathcal{L}$, die die Bedeutung der Ausdrücke darstellt, inklusive von Methodenaufrufen; und (c) eine Variable $workset : N^{am} \times [\mathcal{L} \to \mathcal{L}]$, deren zweite Komponente eine neue Approximation des $[\![\ ]\!]$-Funktional für den Zustand in der ersten Komponente darstellt. In den folgenden Beschreibungen wird jeweils das OO-Analyse-Modell eines Programms verwendet.

Beschreibung der Berechnung von Methodenaufrufen

1. Initialisiere die Transformationen *gtr* für alle Zustände und *ltr* für alle Transitionen mit $\top_{[\mathcal{L}\to\mathcal{L}]}$.

2. Initialisiere die *workset* mit allen Startzuständen der Methoden und der Identitätsfunktion.

3. Berechne iterativ, solange die *workset* nichtleer ist:

   (a) Entnehme der *workset* ein Element. Dies ist das momentan zu analysierende Element.

   (b) Dann wird die bestehende Datenflußinformation, abgelegt in *gtr*, geschnitten (meet) mit der neu berechneten Information, abgelegt in der zweiten Komponente des momentanen Elementes der *workset*.

   (c) Ergibt sich durch den Schnitt eine bessere (genauere) Information[3], so fahre fort, ansonsten beginne wieder mit 3, d.h. die neu berechnete Information ist nicht besser als die alte Information.

   (d) Die alte in *gtr* abgelegte Information wird ersetzt durch die neue Information.

   (e) Falls der betrachtete Zustand (die erste Komponente des momentanen Elementes der *workset*), kein Endzustand einer Methode ist, so wird die *workset* erweitert um die Elemente, bestehend aus den nachfolgenden

[3]Was unter einer besseren (genaueren) Information zu verstehen ist, hängt von dem konkreten Datenflußanalyseproblem ab, das an dieser Stelle aber noch nicht formuliert ist. Hier gilt also: die neue Information ist (verbandstheoretisch) echt enthalten in der alten Information.

Zuständen sowie den Informationen, die sich berechnen aus der Auswertung der lokalen Transformation *ltr* der entsprechenden Transition angewandt auf die neue Information aus 3d. Danach fahre fort mit 3.

(f) Falls der betrachtete Zustand ein Endzustand einer Methode ist, dann führe für alle Aufruf-Transitionen dieser Methode aus:

i. Ersetze die lokale Transformation *ltr* durch den Schnitt der Berechnungen der Aufruf-Filter, der Parameterbindungen und der globalen Transformation *gtr* angewandt auf den Endzustand der entsprechenden gleichnamigen Methoden sowie dem Ausnahmefall *SthOutOf*.

ii. Erweitere die *workset* um das Element bestehend aus dem dirket nachfolgenden Zustand und der Information, die sich berechnet aus der Auswertung der lokalen Transformation *ltr* der Aufruf-Transition aus 3f, angewandt auf die an dieser Transition vorliegenden Information in *gtr*.

(g) Fahre fort mit 3.

Bislang haben wir die Berechnung der abstrakten Semantik von Methoden beschrieben. Nun kommen wir zur vollständigen Analyse eines Programms, wobei die Datenflußinformationen in einem Feld $dfi : N^{am} \to \mathcal{L}$ abgelegt sind und die *workset* aus noch zu analysierenden Zuständen besteht.

3.5.1 Beschreibung der Berechnung der Datenflußinformationen für das gesamte Programm

1. Initialisiere das Feld *dfi* mit $\top$.

2. Initialisiere das *dfi*-Feld des Startzustands des Hauptprogramms mit der Startinformation des Verbandes.

3. Initialisiere die *workset* mit dem Startzustand des Hauptprogramms.

4. Solange die *workset* nichtleer ist, führe aus:

   (a) Entnehme der *workset* ein Element, sei m dieser Zustand.

(b) Führe für alle Transitionen e, deren Anfang gleich m ist, aus:

i. Schneide (meet) die in *dfi* an der Stelle des Zielzustandes der Transition e abgelegte Information mit der lokalen Transformation der Transition e angewandt auf die in *dfi* an der Stelle des Anfangszustandes der Transition e abgelegte Information.

ii. Falls durch 4(b)i eine neue bessere (genauere) Information berechnet worden ist, so fahre fort, ansonsten weiter mit 4b.

iii. Ersetze die *dfi*-Information an dem Zielzustand der Transition e mit der neuen Information. Erweitere die *workset* um diesen Zielzustand.

iv. Falls die momentan bearbeitete Transition e einen Aufruf darstellt, dann fahre fort, ansonsten weiter mit 4b.

v. Für alle Startzustände s der in e aufgerufenen Methode führe aus:

A. Schneide die im *dfi*-Feld des Startzustandes abgelegte Datenflußinformation mit der Berechnung der entsprechenden Aufruf-Filter und der Parameterbindungen angewandt auf die Datenflußinformation abgelegt im *dfi*-Feld an dem Anfangszustand von e.

B. Falls durch 4(b)vA eine bessere Information berechnet worden ist, dann fahre fort, ansonsten weiter mit 4(b)v.

C. Ersetze die Information abgelegt im *dfi*-Feld am Startzustand durch die neue Information.

D. Erweitere die *workset* um den eben betrachteten Startzustand. Fahre fort mit 4b.

Wie bereits erwähnt, läßt sich für die Fixpunktlösung, wie bei intra- und interprozeduralen Datenflußanalysen, ein effektiver Algorithmus angeben, der *OMFP* berechnet. Diesen werden wir nun nach der informellen Beschreibung in einer abstrakten algorithmischen Beschreibung vorstellen. Es wird sich dann zeigen, daß der Algorithmus effektiv ist, d.h. nachweislich terminiert, wenn

1. der Funktionenverband $[\mathcal{L} \to \mathcal{L}]$ die absteigende Kettenbedingung erfüllt,

2. und die lokalen semantischen Funktionale monoton sind.

Algorithmus 3.5.1 *Der Präprozeß: Berechnung der einzelnen Methoden*
Eingabe:

- Ein OO-Analyse-Modell T^{am},
- ein vollständiger Verband $\mathcal{L}$, dessen Funktionenverband $[\mathcal{L} \to \mathcal{L}]$ die absteigende Kettenbedingung erfüllt,
- für jede Transition $e \in E^{ag}$ des zu T^{am} gehörigen OO-Analyse-Graphen T^{ag} die monotone lokale semantische Funktion $[\![e]\!]^{loc}$.

Ausgabe: Eine Annotierung von T^{am} mit Funktionen

- $[\![n]\!] : \mathcal{L} \to \mathcal{L}$ dargestellt durch *gtr* (*global transformation*),
- und $[\![e]\!] : \mathcal{L} \to \mathcal{L}$ dargestellt durch *ltr* (*local transformation*),

die die größte Lösung des Gleichungssystems von Definition 3.4.1 repräsentieren.

Anmerkung: Die Variable *workset* steuert den iterativen Berechnungsprozeß. Die Elemente von *workset* sind Paare, bestehend aus Zuständen $m \in N$ des OO-Analyse-Modells T^{am} und Funktionen $f : \mathcal{L} \to \mathcal{L}$, die eine neue Näherung (Approximation) von $[\![m]\!]$ darstellen. Die Variable *meet* speichert das Ergebnis der jeweils letzten Berechnung von *meet*.

Programm:
(Initialisierung der Annotierungs-Arrays *gtr* und *ltr* sowie der Variablen *workset*)
FORALL $n \in N$ **DO** $gtr[n] := \top_{[\mathcal{L}\to\mathcal{L}]}$ **OD**;
FORALL $e \in E$ **DO**
  IF $e \in E^{am}_{assmc}$
    THEN $ltr[e] := [\![e_{SthOutOf}]\!]^{loc} \sqcap$
      $\bigsqcap \{ [\![e^C_{ass}]\!]^{loc} \circ [\![end(Model(Method(e), C))]\!] \circ$
        $[\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc} \mid C \in Impl(Method(e)) \}$
  ELSEIF $e \in E^{am}_{pmc}$
    THEN $ltr[e] := [\![e_{SthOutOf}]\!]^{loc} \sqcap$
      $\bigsqcap \{ [\![end(Model(Method(e), C))]\!] \circ [\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc}$
        $\mid C \in Impl(Method(e)) \}$
    ELSE $ltr[e] := [\![e]\!]^{loc}$ **FI**
OD;

$workset := \{ (s, \mathsf{Id}_{\mathcal{L}}) \mid s \in start(\mathcal{M}_{Mdl}) \};$

(Iterative Fixpunkt-Berechnung)

WHILE $workset \neq \emptyset$ **DO**

LET $(m, f) \in workset$

BEGIN

$workset := workset \backslash \{ (m, f) \};\ meet := gtr[m] \sqcap f$

IF $gtr[m] \sqsupset meet$

THEN

$gtr[m] := meet;$

IF $m \in \{end(\mathcal{M}_{Mdl})\}$

THEN

FORALL $e \in Sender(mm(m))$ **DO**

IF $e \in E^{am}_{assmc}$

THEN

$$ltr[e] := [\![e_{SthOutOf}]\!]^{loc} \sqcap \bigsqcap \{ [\![e^C_{ass}]\!]^{loc} \circ [\![end(Model(Method(e), C))]\!] \circ [\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc} \mid C \in Impl(Method(e)) \}$$

ELSE

$$ltr[e] := [\![e_{SthOutOf}]\!]^{loc} \sqcap \bigsqcap \{ [\![end(Model(Method(e), C))]\!] \circ [\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc} \mid C \in Impl(Method(e)) \}$$

FI;

$workset := workset \cup \{ (dest(e), ltr[e] \circ gtr[source(e)]) \}$ **OD**

ELSE

$workset := workset \cup$

$\{ (dest(e), ltr[e] \circ gtr[m]) \mid e \in E \text{ with } source(e) = m \}$

FI

FI

END

OD.

Algorithmus 3.5.2 *Berechnung der OMFP-Lösung*

Eingabe:

- Ein OO-Analyse-Modell T^{am},
- ein vollständiger Verband $\mathcal{L}$, der die absteigende Kettenbedingung erfüllt,
- eine Startinformation $c_0 \in \mathcal{L}$,
- für jede Transition $e \in E^{am}$ die monotone lokale semantische Funktion $[\![e]\!]$, berechnet durch Algorithmus 3.5.1,
- für jede Transition $e \in E^{am}_{mc}$ und jede Klasse $C \in \mathit{Impl}(\mathit{Method}(e))$ die lokalen semantischen Funktionen $[\![e']\!]^{loc}$, $e' \in \{e^C_{pb}, e^C_{fil}, e_{SthOutOf}\}$,
- und falls $e \in E^{am}_{assmc}$ auch die lokalen semantischen Funktion $[\![e^C_{ass}]\!]^{loc}$.

Ausgabe: Eine Annotation von T^{am} mit der *OMFP*-Lösung abgelegt in dfi.

Anmerkung: Die Variable *workset* steuert den iterativen Berechnungsprozeß. Die Elemente von *workset* sind Zustände in T^{ag}. Die Variable *meet* speichert das Ergebnis der jeweils letzten Berechnung von *meet*.

Programm:

(Initialisierung des Arrays *dfi* und der Variablen *workset*)
FORALL $n \in N \backslash \{\mathit{start}(M_H)\}$ **DO** $dfi[n] := \top$ **OD**;
$dfi[\mathit{start}(M_H)] := c_0$;
$\mathit{workset} := \{\, \mathit{start}(M_H) \,\}$;
(Iterative Fixpunkt-Berechnung)
WHILE $\mathit{workset} \neq \emptyset$ **DO**
  LET $m \in \mathit{workset}$
    BEGIN
      $\mathit{workset} := \mathit{workset} \backslash \{\, m \,\}$;
      (Neuberechnung der Datenflußinformation des Zustandes m)
      FORALL $e \in E$ with $\mathit{source}(e) = m$ **DO**
        $\mathit{meet} := dfi[\mathit{dest}(e)] \sqcap [\![e]\!](dfi[\mathit{source}(e)])$;
        IF $dfi[\mathit{dest}(e)] \sqsupset \mathit{meet}$
          THEN $dfi[\mathit{dest}(e)] := \mathit{meet}$; $\mathit{workset} := \mathit{workset} \cup \{\, \mathit{dest}(e) \,\}$ **FI**;
        IF $e \in E^{am}_{mc}$
          THEN

FORALL $s \in \{start(Model(Method(e), C)) \mid C \in Impl(Method(e))\}$
DO
$meet := dfi[s] \sqcap [\![e_{pb}^{Class(s)}]\!]^{loc} \circ [\![e_{fil}^{Class(s)}]\!]^{loc}(dfi[source(e)])$;
IF $dfi[s] \sqsupset meet$
THEN $dfi[s] := meet$; $workset := workset \cup \{ s \}$ **FI**
OD
FI
OD
END
OD.

3.6 Korrektheit des *OMFP*-Ansatzes

In diesem Abschnitt werden wir die Korrektheit des *OMFP*-Ansatzes bezüglich des *OMOP*-Ansatzes zeigen.

Lemma 3.6.1
Algorithmus 3.5.1 berechnet die größte Lösung des Gleichungssystems 3.4.1:

$$\forall e \in E^{am} : [\![e]\!] = \bigsqcap \{ ltr^k[e] | k \geq 0 \} \text{ und}$$

$$\forall n \in N^{am} : [\![n]\!] = \bigsqcap \{ gtr^k[n] | k \geq 0 \},$$

wobei ltr^k und gtr^k die Inhalte der Variablen ltr und gtr nach dem k-ten Durchlauf der **WHILE**-Schleife im Algorithums 3.5.1 darstellen. Nach Terminierung des Algorithmuses 3.5.1 gilt:

$$\forall e \in E^{am} : [\![e]\!] = ltr[e] \text{ und}$$

$$\forall n \in N^{am} : [\![n]\!] = gtr[n].$$

Beweis: Seien *fix-gtr* und *fix-ltr* beliebige Lösungen des Gleichungssystems 3.4.1 für $[\![\;]\!]$ und $[\;]$. Zunächst beweisen wir fünf Invarianten des Algorithmuses:

1. $\forall\, k \in \mathbb{N}\ \forall n \in N^{am} :$ *fix-gtr*$[n] \sqsubseteq gtr^k[n]$.

2. $\forall\ k \in \mathbb{N}\ \forall e \in E^{am}:\ \mathit{fix}\text{-}ltr[e] \sqsubseteq ltr^k[e]$.

3. $\forall\ (m,f) \in workset^k : \mathit{fix}\text{-}gtr[m] \sqsubseteq f$.

4. $\forall\ k \in \mathbb{N}\ \forall n \in N^{am}$ mit n ist ein Zustand, der in $workset^k$ vorkommt:

$$gtr^k[n] \sqcap \bigsqcap\{f \mid (n,f) \in workset^k\} =$$

$$\begin{cases} \mathsf{Id}_{\mathcal{L}} & \text{falls } n \in start(\mathcal{M}_{Mdl}) \\ \bigsqcap\{ltr^k[(m,n)] \circ gtr^k[m] \mid m \in pred_{mm(n)}(n)\} & \text{sonst} \end{cases}$$

5. $\forall k \in \mathbb{N}\ \forall e \in E^{am}:\ ltr^k[e] =$

$$\begin{cases} [\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\ \bigsqcap\{\, [\![\, e^C_{ass} \,]\!]^{loc} \circ gtr^k[end(Model(Method(e),C))] \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \\ \qquad\qquad \mid\ C \in Impl(Method(e))\,\} \\ \qquad\qquad \text{falls } e \in E^{am}_{assmc} \\ [\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\ \bigsqcap\{\, gtr^k[end(Model(Method(e),C))] \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \\ \qquad\qquad \mid\ C \in Impl(Method(e))\,\} \\ \qquad\qquad \text{falls } e \in E^{am}_{pmc} \\ [\![\, e \,]\!]^{loc} \qquad\qquad \text{sonst} \end{cases}$$

Die Gültigkeit der Invarianten zeigen wir nun durch Induktion über die Anzahl der Durchläufe der **WHILE**-Schleife aus dem Algorithmus 3.5.1.
$\underline{k=0:}$ Mit der Initialisierung des Algorithmuses erhält man:

$$(a)\quad \forall n \in N^{am}:\ gtr^0[n] = \top_{[\mathcal{L}\to\mathcal{L}]}$$

$$(b)\quad \forall e \in E^{am}:\ ltr^0[e] = \begin{cases} \top_{[\mathcal{L}\to\mathcal{L}]} & \text{falls } e \in E^{am}_{mc} \\ [\![\, e \,]\!]^{loc} & \text{sonst} \end{cases}$$

$$(c)\quad workset^0 = \{(s, \mathsf{Id}_{\mathcal{L}}) \mid s \in \mathcal{M}_{Mdl}\}$$

Also gilt Invariante (1) wegen (a) und Invariante (2) wegen (b); *workset* enthält die Startzustände aller Methodenmodelle und für die gilt nach Definition des Gleichungssystem 3.4.1: $\mathit{fix}\text{-}gtr[n] = \mathsf{Id}_{\mathcal{L}}$, womit auch Invariante (3) erfüllt ist; Invariante (4) ist wegen $gtr^0[n] \sqcap \mathsf{Id}_{\mathcal{L}} = \mathsf{Id}_{\mathcal{L}}$ erfüllt; Invariante (5) folgt direkt aus (b).

$\underline{k-1 \to k:}$ Sei $k > 0$ und (m, f) das im k-ten Schleifendurchlauf bearbeitete Element der *workset*. Der Algorithmus liefert:

$$(d) \quad gtr^k[n] = \begin{cases} gtr^{k-1}[n] \sqcap f & \text{falls } n = m \text{ und } gtr^{k-1}[n] \sqsupset gtr^{k-1}[n] \sqcap f \\ gtr^{k-1}[n] & \text{sonst} \end{cases}$$

$$(e) \quad ltr^k[e] = \begin{cases} [\![e_{SthOutOf}]\!]^{loc} \sqcap \\ \bigsqcap\{ [\![e^C_{ass}]\!]^{loc} \circ gtr^k[end(Model(Method(e), C))] \circ [\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \; C \in Impl(Method(e)) \} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{falls } e \in Sender(mm(m)) \text{ und} \\ \qquad\qquad\qquad\qquad\qquad\qquad m \in end(\mathcal{M}_{Mdl}) \text{ und } e \in E^{am}_{assmc} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{und } gtr^{k-1}[m] \sqsupset gtr^k[m] \\ [\![e_{SthOutOf}]\!]^{loc} \sqcap \\ \bigsqcap\{ gtr^k[end(Model(Method(e), C))] \circ [\![e^C_{pb}]\!]^{loc} \circ [\![e^C_{fil}]\!]^{loc} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \; C \in Impl(Method(e)) \} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{falls } e \in Sender(mm(m)) \text{ und} \\ \qquad\qquad\qquad\qquad\qquad\qquad m \in end(\mathcal{M}_{Mdl}) \text{ und } e \in E^{am}_{pmc} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{und } gtr^{k-1[m]} \sqsupset gtr^k[m] \\ ltr^{k-1}[e] \qquad\qquad\qquad\qquad\qquad\qquad \text{sonst} \end{cases}$$

Nach Induktionsvoraussetzung gilt:

(i) $\forall n \in N^{am} :\ \mathit{fix}\text{-}gtr[n] \sqsubseteq gtr^{k-1}[n]$

(ii) $\forall e \in E^{am} :\ \mathit{fix}\text{-}ltr[e] \sqsubseteq ltr^{k-1}[e]$

(iii) $\mathit{fix}\text{-}gtr[m] \sqsubseteq f$

$$(iv)\quad ltr^{k-1}[e] = \begin{cases} [\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap & \\ \bigsqcap\{\, [\![\, e^C_{ass} \,]\!]^{loc} \circ gtr^{k-1}[end(Model(Method(e), C))] & \\ \qquad \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \mid\ C \in Impl(Method(e)) \,\} & \\ \qquad\qquad \text{falls } e \in E^{am}_{assmc} & \\ [\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap & \\ \bigsqcap\{\, gtr^{k-1}[end(Model(Method(e), C))] \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} & \\ \qquad\qquad \mid\ C \in Impl(Method(e)) \,\} & \\ \qquad\qquad \text{falls } e \in E^{am}_{pmc} & \\ [\![\, e \,]\!]^{loc} \qquad\qquad \text{sonst} & \end{cases}$$

Die Invarianten (1), (2) und (5) folgen hiermit direkt. Um die Gültigkeit der Invarianten (3) und (4) zu zeigen, nehmen wir o.B.d.A. an, daß

$$gtr^k[m] = gtr^{k-1}[m] \sqcap f \sqsubseteq gtr^{k-1}[m].$$

In dem anderen Fall ist die Induktionsbehauptung trivialerweise erfüllt. Mit (d) und $m \notin end(\mathcal{M}_{Mdl})$ erhalten wir:

$$\begin{aligned} & gtr^k[m] \sqcap \bigsqcap\{f' \mid (m, f') \in workset^k\} \\ =\ & gtr^{k-1}[m] \sqcap f \sqcap \bigsqcap\{f' \mid (m, f') \in workset^k\} \\ =\ & gtr^{k-1}[m] \sqcap \bigsqcap\{f' \mid (m, f') \in workset^{k-1}\},\ \text{da } f \in \bigsqcap\{f' \mid ...\} \\ =\ & \bigsqcap\{ltr^{k-1}[(l, m)] \circ gtr^{k-1}[l] \mid l \in pred_{mm(m)}(m)\},\ \text{nach } (IV) \\ =\ & \bigsqcap\{ltr^{k}[(l, m)] \circ gtr^{k}[l] \mid l \in pred_{mm(m)}(m)\} \end{aligned}$$

womit Invariante (4) für $m \notin end(\mathcal{M}_{Mdl})$ bewiesen ist. Für $m \in end(\mathcal{M}_{Mdl})$ folgt aus der Induktionsvoraussetzung die Behauptung, da für $n \in \bigcup\{dest(e) \mid e \in Sender(mm(m))\}$ oder $n \in succ_{mm(m)}(m)$ die *workset*-Variable erneuert wird.

Für die Invariante (3) unterscheiden wir wieder, ob m ein Endzustand eines Methodenmodells ist oder nicht. Sei $m \notin end(\mathcal{M}_{Mdl})$, dann muß gezeigt werden, daß

$$\forall n \in succ_{mm(m)}(m) : \; \mathit{fix\text{-}gtr}[n] \sqsubseteq ltr^k[(m,n)] \circ gtr^k[m],$$

dies gilt aber wegen

$$\forall n \in succ_{mm(m)}(m) : \; \mathit{fix\text{-}gtr}[n] \sqsubseteq \mathit{fix\text{-}ltr}[(m,n)] \circ \mathit{fix\text{-}gtr}[m].$$

Der andere Fall ergibt sich mit ähnlicher Argumentation. Sei $m \in end(\mathcal{M}_{Mdl})$, dann muß gezeigt werden, daß

$$\forall n \in Sender(mm(m)) : \; \mathit{fix\text{-}gtr}[n] \sqsubseteq ltr^k[(m,n)] \circ gtr^k[m],$$

dies gilt aber wegen

$$\forall n \in Sender(mm(m)) : \; \mathit{fix\text{-}gtr}[n] \sqsubseteq \mathit{fix\text{-}ltr}[(m,n)] \circ \mathit{fix\text{-}gtr}[m].$$

Somit folgt aus (1) und (2) die Invariante (3).

Die Behauptung des Lemmas folgt aus den beiden Gleichungen

$$\bigsqcap\{gtr^k[n] \mid k \geq 0\} = \bigsqcap\{f \mid (n,f) \in workset^k, k \geq 0\}$$

und

$$\bigsqcap\{ltr^k[e] \mid k \geq 0\} = \begin{cases} \bigsqcap\{[\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\ \bigsqcap\{\, [\![\, e^C_{ass} \,]\!]^{loc} \circ gtr^k[end(Model(Method(e),C))] \circ \\ \qquad\qquad [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \\ \qquad \mid \; C \in Impl(Method(e)) \,\} \mid k \geq 0\} \\ \qquad\qquad \text{falls } \; e \in E^{am}_{assmc} \\ \bigsqcap\{[\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\ \bigsqcap\{\, gtr^k[end(Model(Method(e),C))] \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \\ \qquad \mid \; C \in Impl(Method(e)) \,\} \mid k \geq 0\} \\ \qquad\qquad \text{falls } \; e \in E^{am}_{pmc} \\ [\![\, e \,]\!]^{loc} \qquad\qquad \text{sonst} \end{cases}$$

und den zuvor bewiesenen Invarianten.

Lemma 3.6.2

Algorithmus 3.5.2 berechnet die größte Lösung des Gleichungssystems 3.4.2:

$$\forall n \in N^{am}: \ OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(n) = \bigsqcap \{ dfi^k{}_{c_0}(n) \mid k \geq 0 \},$$

wobei dfi^k der Inhalt der Variablen *dfi* nach dem k-ten Durchlauf der **WHILE**-Schleife im Algorithums 3.5.2 darstellt. Nach Terminierung des Algorithmuses 3.5.2 gilt:

$$\forall n \in N^{am}: \ OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(n) = dfi_{c_0}(n).$$

Beweis: Sei *fix-dfi* eine beliebige Lösung des Gleichungssystems 3.4.2 für *dfi*. Zunächst beweisen wir drei Invarianten des Algorithmuses:

1. $\forall k \in \mathbb{N}\ \forall n \in N^{am}: \ fix\text{-}dfi[n] \sqsubseteq dfi^k[n]$

2. $\forall k \in \mathbb{N}\ \forall n \in N^{am} \backslash \{start(M_H)\}:$

   $dfi^k[n] \sqcap \bigsqcap \{ \llbracket\ e\ \rrbracket(dfi^k[source(e)]) \mid dest(e) = n \in workset^k \}$

   $$= \begin{cases} \bigsqcap \{ \llbracket\ e_{pb}^{Class(n)}\ \rrbracket^{loc} \circ \llbracket\ e_{fil}^{Class(n)}\ \rrbracket^{loc}(dfi[source(e)]) \mid e \in Sender(mm(n)) \} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{falls } n \in start(\mathcal{M}_{Mdl}) \\ \bigsqcap \{ \llbracket\ (m,n)\ \rrbracket(dfi[m]) \mid m \in pred_{mm(n)}(n) \} \quad \text{sonst} \end{cases}$$

3. $\forall m \in workset^k:$

   $$fix\text{-}dfi[m] \sqsubseteq \begin{cases} \bigsqcap \{ \llbracket\ e_{pb}^{Class(m)}\ \rrbracket^{loc} \circ \llbracket\ e_{fil}^{Class(m)}\ \rrbracket^{loc}(dfi^k[source(e)]) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid e \in Sender(mm(m)) \} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{falls } m \in start(\mathcal{M}_{Mdl}) \backslash \{start(M_H)\} \\ \bigsqcap \{ \llbracket\ (n,m)\ \rrbracket(dfi^k[n]) \mid n \in pred_{mm(m)}(m) \} \quad \text{sonst} \end{cases}$$

Die Gültigkeit der Invarianten zeigen wir nun durch Induktion über die Anzahl der Durchläufe der **WHILE**-Schleife aus dem Algorithmus 3.5.2.
$\underline{k = 0:}$ Mit der Initialisierung des Algorithmuses erhält man sofort die Behauptung.
$\underline{k - 1 \to k:}$ Sei $k > 0$ und m das im k-ten Schleifendurchlauf bearbeitete Element der *workset*. Der Algorithmus liefert:

$$(i)\quad dfi^k[n] = \begin{cases} dfi^{k-1}[n] \sqcap [\![\,(m,n)\,]\!](dfi^{k-1}[m]) \\ \qquad\qquad \text{falls } (m,n) \in E^{am} \\ \qquad\qquad \text{und } dfi^{k-1}[n] \sqsupset dfi^{k-1}[n] \sqcap [\![\,(m,n)\,]\!](dfi^{k-1}[m]) \\ dfi^{k-1}[n] \sqcap [\![\, e_{pb}^{Class(n)} \,]\!]^{loc} \circ [\![\, e_{fil}^{Class(n)} \,]\!]^{loc}(dfi^{k-1}[m]) \\ \qquad\qquad \text{falls } (m,l) \in E_{mc}^{am} \text{ und } n \in start(mm(Method((m,l)))) \\ \qquad\qquad \text{und } dfi^{k-1}[n] \sqsupset\ dfi^{k-1}[n] \sqcap \\ \qquad\qquad\qquad\qquad [\![\, e_{pb}^{Class(n)} \,]\!]^{loc} \circ [\![\, e_{fil}^{Class(n)} \,]\!]^{loc}(dfi^{k-1}[m]) \end{cases}$$

Weiterhin gilt wegen der Induktionsvoraussetzung:

$(ii)\quad \forall k \in \mathbb{N}\ \forall n \in N^{am} :\ \mathit{fix}\text{-}dfi[n] \sqsubseteq dfi^{k-1}[n]$

$$(iii)\quad \mathit{fix}\text{-}dfi[m] \sqsubseteq \begin{cases} \bigsqcap \{\, [\![\, e_{pb}^{Class(m)} \,]\!]^{loc} \circ [\![\, e_{fil}^{Class(m)} \,]\!]^{loc}(dfi^{k-1}[source(e)]) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid e \in Sender(mm(m)) \,\} \\ \qquad\qquad\qquad\qquad \text{falls } m \in start(\mathcal{M}_{Mdl}) \backslash \{start(M_H)\} \\ \bigsqcap \{\, [\![\,(n,m)\,]\!](dfi^{k-1}[n]) \mid n \in pred_{mm(m)}(m) \,\} \quad \text{sonst} \end{cases}$$

Aus den Gleichungen (i)-(iii) und, da m der im k-ten Durchlauf bearbeitete Knoten ist, folgt die Behauptung.

Bei der intraprozeduralen Datenflußanalyse garantieren die Theoreme (Safety and Coincidence Theorem) von Kam und Ullman [KU77] die Korrektheit bzw. die Koinzidenz der MFP-Lösung bzgl. der MOP-Lösung, wenn die lokalen Semantik-Funktionen monoton bzw. distributiv sind. Daß diese Aussage auch auf interprozedurale Datenflußanalysen übertragen werden kann, hat Knoop in [Kn93] bewiesen. Wir stellen nun eine objekt–orientierte Variante des interprozeduralen Falles vor.

Zunächst gehen wir auf die Effekte des Präprozesses ein.

Definition 3.6.3

Im folgenden sei $omop(n)$ definiert durch

$$omop(n) = \begin{cases} \mathsf{Id}_{\mathcal{L}} & \text{falls } n \in start(\mathcal{M}_{Mdl}) \\ \bigsqcap \{\, [\![\, p \,]\!]^*(c_0) \mid p \in COP[start(mm(n)), n] \,\} & \text{sonst} \end{cases}$$

Lemma 3.6.4

Für alle $n \in N^{am}$ gilt:

1. Falls alle semantischen Funktionen $[\![e]\!]^{loc},\ e \in E^{ag}$, monoton sind, dann gilt: $[\![n]\!] \sqsubseteq omop(n)$.

2. Falls alle semantischen Funktionen $[\![e]\!]^{loc},\ e \in E^{ag}$, distributiv sind, dann gilt: $[\![n]\!] = omop(n)$.

Beweis:

1. Wir beweisen mittels Induktion über die Länge des Pfades p

$$(*)\ \forall n \in N^{am}\ \forall p \in COP[start(mm(n)), n] :\ [\![n]\!] \sqsubseteq [\![p]\!]^*,$$

womit der erste Teil des Lemmas direkt folgt.

$\underline{k=0:}$ Pfade der Länge 0 bestehen nur aus einem Startknoten, für den $(*)$ trivialerweise erfüllt ist.

$\underline{k-1 \to k:}$ Da vollständige Pfade, die auch in einem Startknoten enden, die Länge 0 haben, genügt es im folgenden $n \in N^{am} \backslash start(\mathcal{M}_{Mdl})$ zu betrachten, für die es einen Pfad $p \in COP[start(mm(n)), n]$ gibt mit $0 < length(p) \leq k$. Wir zeigen, daß

$$[\![p]\!]^* \sqsupseteq [\![n]\!]$$

gilt unter der Induktionsvoraussetzung

$$(IV)\ \forall m \in N^{am}\ \forall \hat{p} \in COP[start(mm(n)), m] :\ 0 \leq length(\hat{p}) < k :$$
$$[\![\hat{p}]\!]^* \sqsupseteq [\![m]\!]$$

Der Pfad p läßt sich in $p = p'; (m, n)$ zerlegen mit $m \in pred^{ag}(n)$[4] und $p' \in OP[start(mm(n)), m]$. Für die Transition (m, n) unterscheiden wir drei Fälle:

(a) $(m, n) \notin (E^{ag}_{call} \cup E^{ag}_{ret})$: Dann gilt $p' \in COP[start(mm(n)), m]$ und weiter:

$$\begin{array}{lcll} [\![p]\!]^* & = & [\![(m, n)]\!]^{loc} \circ [\![p']\!]^* & \\ & = & [\![(m, n)]\!] \circ [\![p']\!]^* & \text{da } (m, n) \notin (E^{ag}_{call} \cup E^{ag}_{ret}) \\ & \sqsupseteq & [\![(m, n)]\!] \circ [\![m]\!] & \text{wegen } (IV) \\ & \sqsupseteq & [\![n]\!] & \text{Definition von } [\![n]\!] \end{array}$$

(b) $(m, n) \in E^{ag}_{call}$ kann nicht vorkommen, da p ein vollständiger Pfad ist.

[4] $pred^{ag}$ liefert die Menge der direkten vorhergehenden Zustände im OO-Analyse-Graph.

(c) $(m,n) \in E^{ag}_{ret}$: Hierbei ist m nicht Vorgänger von n in N^{am}. Der Pfad p wird wieder zerlegt, sodaß die Aufrufstruktur des Programmes reflektiert wird[5]:

$$p = p''; (e^{Class(m)}_{fil}); (e^{Class(m)}_{pb}); (start(mm(m)), ..., end(mm(m))); (e^{Class(m)}_{ret})$$

Sei $l = source(e^{Class(m)}_{fil})$. Es gilt: $p'' \in OP[start(mm(n)), l]$ und $(start(mm(m)), ..., end(mm(m))) \in COP[start(mm(m)), end(mm(m))]$ und weiter ergibt sich

$$\begin{array}{rcl}
[\![p]\!]^* & = & [\![e^{Class(m)}_{ret}]\!]^{loc} \circ [\![(start(mm(m)), ..., end(mm(m)))]\!]^* \\
 & & \circ [\![e^{Class(m)}_{pb}]\!]^{loc} \circ [\![e^{Class(m)}_{fil}]\!]^{loc} \circ [\![p'']\!]^* \\
\text{wegen } (IV) & \sqsupseteq & [\![e^{Class(m)}_{ret}]\!]^{loc} \circ [\!| end(mm(m)) |\!] \circ [\![e^{Class(m)}_{pb}]\!]^{loc} \\
 & & \circ [\![e^{Class(m)}_{fil}]\!]^{loc} \circ [\![p'']\!]^* \\
\text{nach Definition} & \sqsupseteq & [\![(l,n)]\!] \circ [\![p'']\!]^* \\
\text{wegen } (IV) & \sqsupseteq & [\![(l,n)]\!] \circ [\!| l |\!] \\
\text{Def. von } [\!|\ |\!] & \sqsupseteq & [\!| n |\!]
\end{array}$$

sowie für den SthOutOf-Fall:

$$\begin{array}{rcll}
[\![p]\!]^* & = & [\![e_{SthOutOf}]\!]^{loc} \circ [\![p'']\!]^* & \\
 & \sqsupseteq & [\![(l,n)]\!] \circ [\![p'']\!]^* & \text{wegen } (IV) \text{ und nach Definition} \\
 & \sqsupseteq & [\![(l,n)]\!] \circ [\!| l |\!] & \text{wegen } (IV) \\
 & \sqsupseteq & [\!| n |\!] & \text{Def. von } [\!|\ |\!]
\end{array}$$

2. Es bleibt noch die umgekehrte Inklusion zu zeigen, d.h.

$$\forall n \in N^{am} : \ [\!| n |\!] \sqsupseteq omop(n).$$

Dazu beweisen wir

$$\begin{array}{rl}
(**)\ \forall k \geq 0\ \forall n \in N^{am}\ \forall e \in E^{am} : & (I)\ omop(n) \sqsubseteq gtr^k[n] \text{ und} \\
 & (II)\ [\![e]\!] \sqsubseteq ltr^k[e]
\end{array}$$

wobei gtr^k und ltr^k wieder den Inhalt der Variablen gtr und ltr nach dem k-ten Schleifendurchlauf im Algorithmus 3.5.1 darstellen. Die Aussage (I) wird durch Induktion über die Anzahl der Schleifendurchläufe bewiesen.

[5] Rückkehr-Transitionen werden durch e_{ret} dargestellt und sind entweder mit einer Zuweisung dekoriert oder sind nicht dekoriert. Im letzten Fall gelte $[\![e_{ret}]\!]^{loc} =_{df} \mathsf{Id}_{\mathcal{L}}$.

$\underline{k = 0}$ folgt aus der Initialisierung des Algorithmuses.

$\underline{k - 1 \to k :}$ Sei $n \in N^{am}$ der Zustand mit Eintrag (n, f) in *workset*, der im k-ten Durchlauf der Schleife bearbeitet wird.

(a) $n \in start(\mathcal{M}_{Mdl})$: Alle Startknoten werden mit $f = \mathsf{Id}_{\mathcal{L}}$ initialisiert und können später nicht wieder in die *workset* aufgenommen werden, also gilt

$$omop(n) = \mathsf{Id}_{\mathcal{L}} = gtr^k[n].$$

(b) $n \in N^{am} \backslash start(\mathcal{M}_{Mdl})$: Es muß ein $m \in N^{am}$ und ein $k > l \geq 0$ geben, so daß

$$(n, f) = (n, ltr^l[(m, n)] \circ gtr^l[m])$$

im l-ten Schleifendurchlauf in *workset* aufgenommen wurde. Abhängig von der Art der Transition erhalten wir:

(i) $(m, n) \notin E^{am}_{mc}$: Dann gilt:

$$\begin{aligned}
&omop(n) \\
&\quad = \quad \bigsqcap\{[\![\, p \,]\!]^* \mid p \in COP[start(mm(n)), n]\} \\
&\quad \sqsubseteq \quad \bigsqcap\{[\![\, (m, n) \,]\!]^{loc} \circ [\![\, p' \,]\!]^* \mid p' \in COP[start(mm(n)), m]\} \\
&\qquad \text{da } (m, n) \notin E^{am}_{mc} \\
&\quad \sqsubseteq \quad [\![\, (m, n) \,]\!]^{loc} \circ \bigsqcap \{[\![\, p' \,]\!]^* \mid p' \in COP[start(mm(n)), m]\} \\
&\qquad \text{da } [\![\ ]\!]^{loc} \text{ distributiv} \\
&\quad = \quad [\![\, (m, n) \,]\!]^{loc} \circ omop(m), \text{ Definition von } omop \\
&\quad \sqsubseteq \quad [\![\, (m, n) \,]\!]^{loc} \circ gtr^l[m], \ IV \\
&\quad = \quad ltr^l[(m, n)] \circ gtr^l[m] = f, \text{ siehe Algorithmus}
\end{aligned}$$

(ii) $(m,n) \in E^{am}_{mc}$: Dann gilt:

$$
\begin{aligned}
& omop(n) \\
& \quad = \quad \bigsqcap\{[\![\, p \,]\!]^* \mid p \in COP[start(mm(n)), n]\} \\
& \quad \sqsubseteq \quad \bigsqcap\{([\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\
& \qquad [\![\, e^C_{ret} \,]\!]^{loc} \circ [\![\, p' \,]\!]^* \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc}) \circ [\![\, p'' \,]\!]^* \\
& \qquad\quad \mid p' \in COP[start(Model(Method((m,n)), C))), \\
& \qquad\qquad\qquad\qquad end(Model(Method((m,n)), C)))], \\
& \qquad\quad p'' \in COP[start(mm(n)), m],\ C \in Impl(Method(e))\} \\
& \qquad \text{da } (m,n) \in E^{am}_{mc} \\
& \quad \sqsubseteq \quad ([\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\
& \qquad \bigsqcap\{[\![\, e^C_{ret} \,]\!]^{loc} \circ gtr^l[end(Model(Method((m,n), C)))] \circ [\![\, e^C_{pb} \,]\!]^{loc} \\
& \qquad\qquad\qquad \circ [\![\, e^C_{fil} \,]\!]^{loc})\ \circ gtr^l[m] \ \mid \ C \in Impl(Method(e))\}, \\
& \qquad \text{da } [\![\ ]\!] \text{ distributiv und } IV \\
& \quad \sqsubseteq \quad ltr^l[(m,n)] \circ gtr^l[m] = f, \text{ siehe Algorithmus}
\end{aligned}
$$

Damit erhalten wir:

$$omop(n) \sqsubseteq gtr^{k-1}[n] \sqcap f = gtr^k[n].$$

Es bleibt noch $(II)\ [\![\, e \,]\!] \sqsubseteq ltr^k[e],\ e \in E^{am}$ zu zeigen:
Wenn $n \notin end(\mathcal{M}_{Mdl})$, so verändert der Algorithmus die Variable *ltr* nicht, womit die Behauptung nach Induktionsvoraussetzung gilt.
Wenn $n \in end(\mathcal{M}_{Mdl})$, dann gilt:

$$
ltr^k[e] = \begin{cases}
[\![\, e_{SthOutOf} \,]\!]^{loc} \sqcap \\
\bigsqcap\{[\![\, e^C_{ret} \,]\!]^{loc} \circ gtr^k[end(Model(Method(e, C)))] \\
\qquad\qquad \circ [\![\, e^C_{pb} \,]\!]^{loc} \circ [\![\, e^C_{fil} \,]\!]^{loc} \mid C \in Impl(Method(e))\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{falls } e \in Sender(mm(n)) \\
ltr^{k-1}[e] \qquad\qquad\qquad\qquad\qquad\qquad \text{sonst}
\end{cases}
$$

Die Aussage $[\![\, e \,]\!] \sqsubseteq ltr^k[e]$ ist eine direkte Folge von der Definition des Gleichungssystems 3.4.1, von (I) und der Induktionsvoraussetzung.

Als unmittelbare Folgerung aus diesem Lemma erhalten wir die folgende Aussage, die später im Beweis der Hauptaussage (Korrektheit und Koinzidenz der *OMFP*-Lösung bezüglich der *OMOP*-Lösung) zur Anwendung kommen wird.

Lemma 3.6.5

Für alle $e \in E^{am}_{mc}$ gilt:

1. Falls alle semantischen Funktionen $[\![e']\!]^{loc},\ e' \in E^{ag}$, monoton sind, dann ist $[\![e]\!] \sqsubseteq \bigsqcap \{[\![p]\!]^* \mid p \in COP[source(e), dest(e)]\}$.

2. Falls alle semantischen Funktionen $[\![e']\!]^{loc},\ e' \in E^{ag}$, distributiv sind, dann ist $[\![e]\!] = \bigsqcap \{[\![p]\!]^* \mid p \in COP[source(e), dest(e)]\}$.

Wir kommen nun zur ersten Hauptaussage.

Theorem 3.6.6 *(OO-Korrektheitstheorem)*
Sei Π ein SOL -Programm und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann ist die *OMFP*-Lösung eine korrekte Approximation der *OMOP*-Lösung, d.h.

$$\forall\, c_0 \in \mathcal{L}\ \forall\, n \in N^{ag} : OMFP_{([\![ ]\!]^{loc}, c_0)}(n) \sqsubseteq OMOP_{([\![ ]\!]^{loc}, c_0)}(n),$$

wenn alle lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc}$, $e \in E^{ag}$, monoton sind.
Beweis: Wir zeigen für alle $\forall c_0 \in \mathcal{L}$:

$$(*)\ \forall n \in N^{am}\ \forall p \in OP[s^{ag}, n] :\ OMFP_{([\![ ]\!]^{loc}, c_0)}(n) \sqsubseteq [\![p]\!]^*(c_0)$$

durch Induktion über die Länge k des Pfades p.
$\underline{k = 0 :}\ \ [\![p]\!]^*(c_0) = [\![\epsilon]\!]^*(c_0) = c_0 = dfi_{c_0}(s^{ag})$
$\underline{k - 1 \to k :}$ Sei $k > 0$ und für alle Pfade p' der Länge $0 \leq length(p') < k$ gilt die Induktionsvoraussetzung:

$$(IV)\ \ \forall m \in N^{am}\ \forall p' \in OP[s^{ag}, m] :\ OMFP_{([\![ ]\!]^{loc}, c_0)}(m) \sqsubseteq [\![p']\!]^*(c_0).$$

Sei p ein Pfad der Länge k, dann kann p zerlegt werden in $p = p'; (m, n)$ für ein $m \in pred^{ag}(n)$ und $p' \in OP[s^{ag}, m]$. Nach (IV) gilt: $OMFP_{([\![ ]\!]^{loc}, c_0)}(m) \sqsubseteq [\![p']\!]^*(c_0)$. Wir betrachten alle möglichen Transitionen und erhalten:

1. $(m, n) \notin (E^{ag}_{call} \cup E^{ag}_{ret})$:

$$\begin{aligned} OMFP_{([\![ ]\!]^{loc}, c_0)}(n) &= \bigsqcap \{[\![(m', n)]\!](dfi_{c_0}(m')) \mid m' \in pred_{mm(n)}(n)\} \\ &\sqsubseteq [\![(m, n)]\!](dfi_{c_0}(m)) \\ &\sqsubseteq [\![(m, n)]\!]([\![p']\!]^*(c_0)) \ \text{ nach } (IV) \\ &= [\![p]\!]^*(c_0) \end{aligned}$$

2. $(m,n) \in E^{ag}_{call}$: Dann ist $n \in start(\mathcal{M}_{Mdl})$ und p kann weiter in

$$p = p''; (\hat{m}, m); (m, n);$$

zerlegt werden. $(\hat{m}, m)$ entspricht einer Aufruf-Transition. Nach (IV) gilt:

$$OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(\hat{m}) \sqsubseteq \llbracket\ p''\ \rrbracket^*(c_0) \text{ für } p'' \in OP[s^{ag}, \hat{m}].$$

Damit gilt:

$$\begin{aligned}
&OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(n) \\
&\quad = \quad \bigsqcap \{ \llbracket\ e^{Class(n)}_{pb}\ \rrbracket^{loc} \circ \llbracket\ e^{Class(n)}_{fil}\ \rrbracket^{loc}(dfi_{c_0}[source(e)]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \ e \in Sender(mm(n)) \} \\
&\quad \sqsubseteq \quad \llbracket\ e^{Class(n)}_{pb}\ \rrbracket^{loc} \circ \llbracket\ e^{Class(n)}_{fil}\ \rrbracket^{loc}(dfi_{c_0}(\hat{m})) \\
&\quad = \quad \llbracket\ e^{Class(n)}_{pb}\ \rrbracket^{loc} \circ \llbracket\ e^{Class(n)}_{fil}\ \rrbracket^{loc}(OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(\hat{m})) \\
&\qquad \text{nach } (IV) \\
&\quad \sqsubseteq \quad \llbracket\ e^{Class(n)}_{pb}\ \rrbracket^{loc} \circ \llbracket\ e^{Class(n)}_{fil}\ \rrbracket^{loc}(\llbracket\ p''\ \rrbracket^*(c_0)) \\
&\quad = \quad \llbracket\ p\ \rrbracket^*(c_0)
\end{aligned}$$

3. $(m,n) \in E^{ag}_{ret}$: Der Pfad p hat also die Gestalt

$p = p'; (e^{Class(m)}_{fil}); (e^{Class(m)}_{pb}); (start(mm(m)), ..., end(mm(m))); (e^{Class(m)}_{ret})$
bzw.
$p = p'; (e_{SthOutOf}); (dest(e^{Class(m)}_{SthOutOf}), n).$

Wenn wir m' und $source(e^{Class(m)}_{fil})$ bzw. $source(e_{SthOutOf})$ identifizieren, dann erhalten wir aus der Induktionsvoraussetzung:

$$OMFP_{(\llbracket\ \rrbracket^{loc}, c_0)}(m') \sqsubseteq \llbracket\ p'\ \rrbracket^*(c_0) \text{ für } p' \in OP[s^{ag}, m'].$$

Es gilt:

$$
\begin{aligned}
& OMFP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) \\
&\quad = \bigsqcap \{\llbracket\,(\hat{m},n)\,\rrbracket(dfi_{c_0}(\hat{m})) \mid \hat{m} \in pred_{mm(n)}(n)\} \\
&\quad \sqsubseteq \llbracket\,(m',n)\,\rrbracket(dfi_{c_0}(m')) \\
&\quad \sqsubseteq \llbracket\,(m',n)\,\rrbracket(\llbracket\,p'\,\rrbracket^*(c_0)) \\
&\qquad \text{nach } (IV) \\
&\quad \sqsubseteq \bigsqcap \{\llbracket\,\hat{p}\,\rrbracket^*(\llbracket\,p'\,\rrbracket^*(c_0)) \mid \hat{p} \in COP[m',n]\} \\
&\qquad \text{wegen Lemma 3.6.5} \\
&\quad \sqsubseteq (\llbracket\,e_{ret}^{Class(m)}\,\rrbracket^{loc} \circ \llbracket\,(start(mm(m)),...,end(mm(m)))\,\rrbracket^* \circ \\
&\qquad \llbracket\,e_{pb}^{Class(m)}\,\rrbracket^{loc} \circ \llbracket\,e_{fil}^{Class(m)}\,\rrbracket^{loc})(\llbracket\,p'\,\rrbracket^*(c_0)) \\
&\quad = \llbracket\,p\,\rrbracket^*(c_0)
\end{aligned}
$$

bzw.:

$$
\begin{aligned}
OMFP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) &= \bigsqcap \{\llbracket\,(\hat{m},n)\,\rrbracket(dfi_{c_0}(\hat{m})) \mid \hat{m} \in pred_{mm(n)}(n)\} \\
&\sqsubseteq \llbracket\,(m',n)\,\rrbracket(dfi_{c_0}(m')) \\
&\sqsubseteq \llbracket\,(m',n)\,\rrbracket(\llbracket\,p'\,\rrbracket^*(c_0)) \\
&\sqsubseteq \bigsqcap \{\llbracket\,\hat{p}\,\rrbracket^*(\llbracket\,p'\,\rrbracket^*(c_0)) \mid \hat{p} \in COP[m',n]\} \\
&\qquad \text{wegen Lemma 3.6.5} \\
&\sqsubseteq \llbracket\,e_{SthOutOf}\,\rrbracket^{loc}(\llbracket\,p'\,\rrbracket^*(c_0)) \\
&= \llbracket\,p\,\rrbracket^*(c_0).
\end{aligned}
$$

Die *OMFP*-Lösung fällt mit der *OMOP*-Lösung zusammen, wenn zusätzlich noch die Distributivität der lokalen Semantikfunktionen erreicht werden kann.

Theorem 3.6.7 *(OO-Koinzidenztheorem)*
Sei Π ein SOL -Programm und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann ist die *OMFP*-Lösung identisch zur *OMOP*-Lösung, d.h.

$$\forall\, c_0 \in \mathcal{L}\ \forall\, n \in N^{ag} : OMFP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) = OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n),$$

wenn alle lokalen abstrakten Semantikfunktionen $\llbracket\,e\,\rrbracket^{loc}$, $e \in E^{ag}$, distributiv sind.
Beweis: Da distributive Funktionen auch monoton sind, genügt es zu zeigen, daß

$$(*)\ \forall k \geq 0\ \forall n \in N^{am} :\ OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) \sqsubseteq dfi^k[n].$$

Wir beweisen dies durch Induktion über die Anzahl der Schleifendurchläufe im Algorithmus 3.5.2.

$\underline{k=0:}$ Da *dfi* mit $\top$ initialisiert wird, ist die Aussage für den Induktionsanfang korrekt.

$\underline{k=1:}$ Zunächst wird der Startzustand bearbeitet, dies bedeutet:

$$OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(s^{ag}) \sqsubseteq \llbracket\ \epsilon\ \rrbracket^*(c_0) = \mathsf{Id}_{\mathcal{L}} = dfi[s^{ag}].$$

$\underline{k-1 \to k:}$ Sei $n \in N^{am}$ der Zustand der *workset*, der im k-ten Durchlauf der Schleife bearbeitet wird. Es gibt einen Zustand $m \in N^{am}$, dessen Bearbeitung im l-ten Durchlauf der Schleife bewirkt, daß n in die *workset* aufgenommen wurde. Für diesen Zustand liefert uns der Algorithmus folgende Beziehung:

$$sp = \begin{cases} \llbracket\ e_{pb}^{Class(n)}\ \rrbracket^{loc} \circ \llbracket\ e_{fil}^{Class(n)}\ \rrbracket^{loc}(dfi^l[m]) & \text{falls } n \in start(\mathcal{M}_{Mdl}) \\ \llbracket\ (m,n)\ \rrbracket(dfi^l[m]) & \text{sonst} \end{cases}$$

Wir beweisen zunächst $OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) \sqsubseteq sp$:

1. $(m,n) \in E^{am} \backslash E^{am}_{mc}$:
   Hierfür gilt:

$$\begin{array}{lcl} OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) & = & \bigsqcap \{\llbracket\ p\ \rrbracket^*(c_0) \mid p \in OP[s^{ag},n]\} \\ ((m,n) \notin E^{am}_{mc}) & \sqsubseteq & \bigsqcap \{\llbracket\ (m,n)\ \rrbracket^{loc}(\llbracket\ p'\ \rrbracket^*(c_0)) \mid p' \in OP[s^{ag},m]\} \\ (\llbracket\ \rrbracket^{loc}\text{distributiv}) & = & \llbracket\ (m,n)\ \rrbracket^{loc}(\bigsqcap \{\llbracket\ p'\ \rrbracket^*(c_0) \mid p' \in OP[s^{ag},m]\}) \\ (Def.\ OMOP) & = & \llbracket\ (m,n)\ \rrbracket^{loc}(OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(m)) \\ ((m,n) \notin E^{am}_{mc}) & = & \llbracket\ (m,n)\ \rrbracket(OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(m)) \\ (IV) & \sqsubseteq & \llbracket\ (m,n)\ \rrbracket(dfi^l[m]) \\ & = & sp \end{array}$$

2. $(m,n) \in E^{am}_{mc}$:

Hierfür gilt:

$$\begin{array}{lcl}
OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) & = & \bigsqcap\{\llbracket\, p\,\rrbracket^*(c_0)\,|\,p\in OP[s^{ag},n]\}\\
& \sqsubseteq & \bigsqcap\{\llbracket\, p'\,\rrbracket^*(\llbracket\, p''\,\rrbracket^*(c_0))\\
& & \qquad\qquad |\,p''\in OP[s^{ag},m],\ p'\in COP[m,n]\}\\
(\llbracket\ \rrbracket^{loc}\text{distributiv}) & = & \bigsqcap\{\llbracket\, p'\,\rrbracket^*(\bigsqcap\{\llbracket\, p''\,\rrbracket^*(c_0)\,|\,p''\in OP[s^{ag},m]\})\\
& & \qquad\qquad |\ \ p'\in COP[m,n]\}\\
(\text{Def. } OMOP) & = & \bigsqcap\{\llbracket\, p'\,\rrbracket^*(OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(m))\mid\ p'\in COP[m,n]\}\\
(\text{Lemma 3.6.5}) & = & \llbracket\,(m,n)\,\rrbracket(OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(m))\\
(IV) & \sqsubseteq & \llbracket\,(m,n)\,\rrbracket(dfi^l[m])\\
& = & sp
\end{array}$$

3. $(m,n)\notin E^{am}$:

   Dann ist $(m,l)\in Sender(mm(n))$ und $n\in start(\mathcal{M}_{Mdl})$. Wir erhalten:

$$\begin{array}{lcl}
OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n) & = & \bigsqcap\{\llbracket\, p\,\rrbracket^*(c_0)\,|\,p\in OP[s^{ag},n]\}\\
& \sqsubseteq & \bigsqcap\{\llbracket\, e_{pb}^{Class(n)}\,\rrbracket^{loc}\circ\\
& & \qquad\llbracket\, e_{fil}^{Class(n)}\,\rrbracket^{loc}(\llbracket\, p'\,\rrbracket^*(c_0))\,|\,p'\in OP[s^{ag},m]\}\\
(\llbracket\ \rrbracket^{loc}\text{distributiv}) & = & \llbracket\, e_{pb}^{Class(n)}\,\rrbracket^{loc}\circ\\
& & \qquad\llbracket\, e_{fil}^{Class(n)}\,\rrbracket^{loc}(\bigsqcap\{\llbracket\, p'\,\rrbracket^*(c_0)\,|\,p'\in OP[s^{ag},m]\})\\
(\text{Def. } OMOP) & = & \llbracket\, e_{pb}^{Class(n)}\,\rrbracket^{loc}\circ\llbracket\, e_{fil}^{Class(n)}\,\rrbracket^{loc}(OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(m))\\
(IV) & \sqsubseteq & \llbracket\, e_{pb}^{Class(n)}\,\rrbracket^{loc}\circ\llbracket\, e_{fil}^{Class(n)}\,\rrbracket^{loc}(dfi^l[m])\\
& = & sp
\end{array}$$

Insgesamt erhalten wir hieraus:

$$OMOP_{(\llbracket\ \rrbracket^{loc},c_0)}(n)\sqsubseteq dfi^{k-1}[n]\sqcap sp = dfi^k[n].$$

Kapitel 4

Typanalyse

Als Anwendung des im vorigen Kapitel vorgestellten Analyse-Rahmens definieren wir nun Typanalysen für ungetypte objekt-orientierte Programmiersprachen, zunächst als monotone Datenflußanalysen und dann als distributive Datenflußanalyse.

Wir werden insgesamt vier monotone Datenflußanalysen mit unterschiedlichen Mächtigkeiten präsentieren. Die entscheidenden Merkmale der zunächst definierten Typanalyse MONO (als monotone Datenflußanalyse) sind:

- separate Bearbeitung der Methodenaufrufe, d.h. jeder Methodenaufruf wird unabhängig von allen anderen analysiert (in der Literatur auch polyvariante oder polymorphe Behandlung von Methodenaufrufen genannt),
- partiell deterministische Behandlung der Verzweigungsausdrücke, d.h. sofern eindeutige Informationen über den Wert der Bedingung des Verzweigungsausdruckes vorliegen, wird nur der geeignete Zweig analysiert,
- destruktive Behandlung der Zuweisungsausdrücke, d.h. Überschreiben des Wertes der Variablen auf der linken Seite durch den Wert des Ausdrucks auf der rechten Seite des Zuweisungsausdrucks.

Die drei weiteren Typanalysen sind eingeschränkte Versionen der Typanalyse MONO. Bei der Typanalyse MONO-ASS wird die scharfe Behandlung von Zuweisungen aufgehoben, indem nun der bereits zuvor berechnete Typ der Variablen um den neuen Typ erweitert wird (kumulative oder "collecting" Behandlung von Zuweisungen). Die Typanalyse MONO-BRANCH unterscheidet sich von MONO durch eine

nichtdeterministische Behandlung der Verzweigungsausdrücke, d.h. es werden immer beide Zweige analysiert. Bei der vierten monotonen Typanalyse PS werden beide Einschränkungen zur Anwendung kommen. PS entspricht damit von ihrer Mächtigkeit her in etwa der Typinferenz von Palsberg/Schwartzbach [OPS92]. Daher haben wir ihr auch den Namen PS gegeben.

Die aggressivste (präziseste) aller Typanalysen ist die Typanalyse DIST, die als distributives Datenflußanalyseproblem definiert wird. Die Datenflußinformationen an den Programmpunkten werden derart vereint, daß es auch an einem späteren Programmpunkt möglich ist, die zuvor zusammengefaßten Informationen wieder auseinanderhalten zu können.

Die zur Definition einer Typanalyse als Datenflußanalyseproblem wichtigsten Schritte sind:

1. Definition der Datenflußinformationen und des der Datenflußanalyse zugrundeliegenden Verbandes,

2. Definition der lokalen abstrakten Semantikfunktionen,

3. Nachweis, daß der benutzte Verband die absteigende Kettenbedingung erfüllt,

4. Nachweis der Monotonie bzw. Distributivität der lokalen abstrakten Semantikfunktionen.

Nach der Spezifikation und dem Nachweis der Korrekheit der Typanalyse werden wir auch noch auf die Komplexität der Analyse eingehen.

4.1 Die Typanalyse MONO

Zunächst präsentieren wir als monotone Datenflußanalyse die Typanalyse MONO.

4.1.1 Spezifikation der Typanalyse

Sei im folgenden Π ein SOL -Programm und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann definieren wir:

- $\mathcal{C}' =_{df} \mathcal{C}_{Id} \cup \{Int, False, True, Nil\}$ die Menge der Klassenidentifikatoren von Π erweitert um den Typ *Integer* als Repräsentant der Klasse der ganzen Zahlen und um die Typen *False*, *True* und *Nil* als Repräsentanten der speziellen Werte `false`, `true` und `nil`;
- $\mathcal{V}' =_{df} \mathcal{V}_{Id}(T^{ag})$ die Menge der in T^{ag} vorkommenden Variablenidentifikatoren und Argumentidentifikatoren;
- $\mathcal{PS} =_{df} (\mathcal{P}(\mathcal{C}') \cup \{\bot_{ps}, \top_{ps}\}, \sqcap, \sqcup, \bot_{ps}, \top_{ps})$ den Potenzmengenverband von $\mathcal{C}'$ erweitert um ein kleinstes Element $\bot_{ps}$ und ein größtes Element $\top_{ps}$, wobei $\mathcal{P}$ den Potenzmengenoperator bezeichnet.

Datenbereich und Datenflußinformationen

Der Datenbereich der Typanalyse wird definiert als der Verband der Menge der Funktionen

$$(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{FL}, \cup, \cap, \supseteq, \top_{fl}, \bot_{fl})$$

mit

- $\mathcal{FL} =_{df} [\mathcal{V}' \to \mathcal{PS}]$,
- $\cup$, $\cap$, $\supseteq$, die die punktweisen Mengenoperationen Vereinigung, Schnitt und umfassend oder gleich sein darstellen,
- $\forall\, v \in \mathcal{V}' :\ \top_{fl}(v) =_{df} \top_{ps}$,
- $\forall\, v \in \mathcal{V}' :\ \bot_{fl}(v) =_{df} \bot_{ps}$.

Diejenigen Programmpunkte, die nach Terminierung der Typanalyse mit $\bot_{fl}$ annotiert sind, stellen unerreichbaren Programmcode (dead code) dar. Gibt es einen Programmpunkt, der mit $\top_{fl}$ annotiert ist, so bedeutet dies, daß ein Methodenaufruf zur Laufzeit möglicherweise nicht verstanden (ausgeführt) werden kann.

Bevor wir zu der Definition der lokalen Semantikfunktionen kommen, noch eine weitere Notation: $\forall M \in \mathcal{PS}, v \in \mathcal{V}', f \in \mathcal{FL} \backslash \{\top_{fl}, \bot_{fl}\}$: f ist die eindeutige Funktion aus $\mathcal{FL}$ mit

$$f[v \to M](x) =_{df} \begin{cases} M & \text{falls x=v} \\ f(x) & \text{sonst} \end{cases}$$

Lokale abstrakte Semantik

Das Funktional

$$[\![\ ]\!]^{loc} : E^{ag} \to (\mathcal{FL} \to \mathcal{FL})$$

beschreibt die lokale abstrakte Semantik und wird definiert durch:

- *Zuweisungen:* Wenn e mit einer Zuweisung der folgenden Form dekoriert ist, dann
  - $Id := SE$

$$[\![\ e\ ]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \\ f[Id \to \{A\}] & \text{falls } SE \equiv A\ \texttt{new} \\ f[Id \to \{Class(e)\}] & \text{falls } SE \equiv \texttt{self} \\ f[Id \to f(x)] & \text{falls } SE \equiv x,\ x \in \mathcal{V}' \\ f[Id \to \{Nil\}] & \text{falls } SE \equiv \texttt{nil} \\ f[Id \to \{Int\}] & \text{falls } SE \equiv Integer \end{cases}$$

  - $Id := Id_1\ \{+ | - | * | /\}\ Id_2$

$$[\![\ e\ ]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \\ f[Id \to \{Int\}] & \text{falls } f(Id_1) = f(Id_2) = \{Int\} \\ \top_{fl} & \text{sonst} \end{cases}$$

  - $e\text{-}cond := e\text{-}cond\ \texttt{instanceOf}\ C$

$$[\![\ e\ ]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad \{C\} = f(e\text{-}cond) \\ f[e\text{-}cond \to \{False\}] & \text{falls } C \notin f(e\text{-}cond) \\ f[e\text{-}cond \to \{True, False\}] & \text{sonst} \end{cases}$$

  - $e\text{-}cond := Id_1\ \{= | < | > | <= | >= | <>\}\ Id_2$:

$$[\![\ e\ ]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \\ f[e\text{-}cond \to \{False, True\}] & \text{sonst} \end{cases}$$

- *Ausdrücke:* Wenn e dekoriert ist mit einem Ausdruck Form SE: $[\![\ e\ ]\!]^{loc}(f) =_{df} f$

- *Filter:* Wenn e mit einem Filter der folgenden Form dekoriert ist, dann
  - e-$sendsTo\ C.m_1 : \ldots m_n :$

$$[\![e]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad C \in f(e\text{-}sendsTo) \\ \bot_{fl} & \text{sonst} \end{cases}$$

  - e-$sendsTo$ `SthOutOf` $Impl(Method(e))$

$$[\![e]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \\ \top_{fl} & \text{falls } f(e\text{-}sendsTo) \backslash Impl(Method(e)) \neq \emptyset \\ \bot_{fl} & \text{sonst} \end{cases}$$

  - e-$cond$ `eqFalse`

$$[\![e]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad False \in f(e\text{-}cond) \\ \bot_{fl} & \text{sonst} \end{cases}$$

  - e-$cond$ `neqFalse`

$$[\![e]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad f(e\text{-}cond) \backslash \{False\} \neq \emptyset \\ \bot_{fl} & \text{sonst} \end{cases}$$

Startinformation: ist die Funktion $f_0 \in \mathcal{FL}$ mit $f_0(v) = \{\}\ \forall v \in \mathcal{V}'$.

Die Startinformation drückt aus, daß an jedem Programmpunkt noch keine echte Information bekannt ist[1].

Die Mächtigkeit der Typanalyse wird vor allem durch die Filtertransitionen erreicht. Sie ermöglichen eine quasi-deterministische Behandlung von Methodenaufrufen und Verzweigungen. Die Aufgabe eines Filters ist es, Informationen nur

[1] In Smalltalk-80 werden Variablen automatisch bei ihrer Definition mit `nil` initialisiert. Da SOL sich an Smalltalk-80 orientiert, geschieht dies auch in SOL -Programmen. Somit könnte die Typanalyse auch mit der Startinformation $f_0(v) = \{Nil\}\ \forall v \in \mathcal{V}'$ beginnen. Warum wir dies aber nicht verwenden, wird im folgenden Kapitel unter dem Stichwort "Behandlung von uninitialisierten Variablen" diskutiert. Die oben gewählte Startinformation kann als ein Tribut der Theorie an die praktische Anwendung gesehen werden.

dann weiterzuleiten, wenn die durch den Filter ausgedrückten Anforderungen an das Programm erfüllt sind bzw. ausgeführt werden können. Falls dies nicht der Fall ist, dann blockt der Filter die Propagierung der Typinformationen, indem er nur $\bot_{fl}$ weiterreicht. Z.B. propagiert der Filter *e-sendsTo* $C.m_1 : \ldots m_n$: die anliegende Information nur dann weiter, wenn der Empfänger Instanz der Klasse C ist. Entsprechend reicht der Filter *e-sendsTo* `eqFalse` nur dann die anliegende Information weiter, wenn die Bedingung *e-cond* den Wert *False* hat. Der Filter *e-sendsTo* `SthOutOf` *Impl*(*Method*(e)), der den Ausnahmefall *message-does-not-understand* behandelt, gibt entweder die Information $\top_{fl}$ weiter ($\top_{fl}$ ist das kleinste Element des Verbandes $\mathcal{FL}$), wenn ein Methodenaufruf an ein Objekt geschickt worden ist, dessen Klasse keine geeignete Methode enthält; ansonsten wird $\bot_{fl}$ weitergereicht.

Die Definition der Bedeutung von Zuweisungen zeigt, warum wir diese als destruktiv bezeichnet haben. In der Typanalyse wird der Wert der Variablen, die auf der linken Seite einer Zuweisung vorkommt, mit dem Wert des Ausdrucks der rechten Seite überschrieben. In einigen Arbeiten [PS91, Gr95] werden die alten Typinformationen nur um den neuen Wert erweitert; die alten Werte bleiben weiterhin erhalten. Dies ist aber aus Sicht unserer Typanalyse nicht notwendig.

Sprachgebrauch

Eine Typanalyse akzeptiert ein Programm genau dann, wenn es keinen Programmpunkt (Zustand) gibt, der nach Terminierung der Analyse mit $\bot$ ($= \top_{fl}$ bei MONO) annotiert ist. Dies bedeutet, daß die Typanalyse alle Methodenaufrufe und arithmetische Operationen als korrekt anerkennt. Entsprechend sagen wir, daß eine Typanalyse ein Programm ablehnt (rejected), wenn ein Methodenaufruf oder eine arithmetische Operation gefunden wird, die zu einem Fehler führt, d.h. das Ergebnis $\bot$ liefert.

4.1.2 Korrektheit der monotonen Typanalyse

Da der zugrundeliegende Verband $\mathcal{FL}$ endlich ist, erhalten wir sofort die folgende Aussage.

Lemma 4.1.1

$\mathcal{FL}$ ist ein vollständiger Verband, der die absteigende Kettenbedingung erfüllt.

Nach der erfolgreichen Spezifikation der Typanalyse zeigen wir nun die Monotonie der lokalen abstrakten Semantikfunktionen $[\![\cdot]\!]^{loc}$, um somit die Korrektheit sicherzustellen.

Lemma 4.1.2

Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc}$, $e \in E^{ag}$, von MONO sind monoton.

Beweis: Monotonie bedeutet:

$$\forall\, [\![e]\!]^{loc} \in [\mathcal{FL} \to \mathcal{FL}]\ \forall f, f' \in \mathcal{FL} :\ f \sqsubseteq f' \Rightarrow [\![e]\!]^{loc}(f) \sqsubseteq [\![e]\!]^{loc}(f').$$

Im folgenden sei $f \sqsubseteq f'$. Wir beweisen nur für zwei Ausdrucksarten exemplarisch die Aussage des Lemmas. Alle anderen Fälle lassen sich in gleicher Weise zeigen.

1. Sei e mit einer Zuweisung der Form $Id := SE$ dekoriert.

   (a) Sei weiter $f = \top_{fl}$ und f' beliebig. Dann erhalten wir $\top_{fl} \sqsubseteq f'$ und auch $\top_{fl} \sqsubseteq [\![e]\!]^{loc}(f')$, wegen der Verbandseigenschaft von $\mathcal{FL}$. Also $[\![e]\!]^{loc}(f) \sqsubseteq [\![e]\!]^{loc}(f')$. Der Fall $f = \bot_{fl}$ läßt für f' nur die Wahl $f' = \bot_{fl}$ zu, womit wir die Aussage sofort bekommen.

   Sei nun $f' = \bot_{fl}$ und f beliebig. Dann gilt $f \sqsubseteq \bot_{fl}$ und auch $[\![e]\!]^{loc}(f) \sqsubseteq \bot_{fl}$, wegen der Verbandseigenschaft von $\mathcal{FL}$; also $[\![e]\!]^{loc}(f) \sqsubseteq [\![e]\!]^{loc}(f')$. Der Fall $f' = \top_{fl}$ läßt für f nur die Wahl $f = \top_{fl}$ zu. Hierfür ist die Aussage trivialerweise erfüllt.

   (b) Sei nun $x \in \mathcal{V}' \backslash \{Id\}$ beliebig. Es gilt

   $$f(x) \sqsubseteq f'(x) \Rightarrow [\![e]\!]^{loc}(f(x)) \sqsubseteq [\![e]\!]^{loc}(f'(x)),$$

   da sich für diese Variablen bei Anwendung der Semantikfunktionen keine Änderungen ergeben.

   Wir betrachten daher $x \in \{Id\}$ mit $f(Id) \sqsubseteq f'(Id)$. Nach Anwendung der Zuweisung werden in beiden Funktionen f und f' jeweils die Komponente x auf den gleichen Wert gesetzt; also $[\![e]\!]^{loc}(f(x)) = [\![e]\!]^{loc}(f'(x))$.

   Insgesamt erhalten wir: $[\![e]\!]^{loc}(f) \sqsubseteq [\![e]\!]^{loc}(f')$

2. Sei e mit einem Filter der Form *e-sendsTo* $C.m_1 : \ldots m_n$: dekoriert. Wir betrachten nur die "interessanten" Fälle, d.h. $C \in f(\textit{e-sendsTo})$ bzw. $C \notin f(\textit{e-sendsTo})$. Für alle weiteren Fälle gilt mit entsprechender Argumentation wie zuvor: $[\![e]\!]^{loc}(f) \sqsubseteq [\![e]\!]^{loc}(f')$.

   (a) $C \in f(\textit{e-sendsTo})$ und $f(\textit{e-sendsTo}) \sqsubseteq f'(\textit{e-sendsTo})$: Es können für f' zwei Fälle eintreten:

      i. $C \notin f'(\textit{e-sendsTo})$
      $\Rightarrow [\![e]\!]^{loc}(f(\textit{e-sendsTo})) \sqsubseteq [\![e]\!]^{loc}(f'(\textit{e-sendsTo})) = \bot_{fl} \Rightarrow$ Beh.

      ii. $C \in f'(\textit{e-sendsTo}) \Rightarrow [\![e]\!]^{loc}(f(\textit{e-sendsTo})) = [\![e]\!]^{loc}(f'(\textit{e-sendsTo}))$
      $\Rightarrow$ Beh., da die Ausgangsinformation weitergeleitet wird und für alle $x \in \mathcal{V}' \backslash \{\textit{e-sendsTo}\}$ unverändert $[\![e]\!]^{loc}(f(x)) \sqsubseteq [\![e]\!]^{loc}(f'(x))$ gilt.

   (b) $C \notin f(\textit{e-sendsTo})$ und $f(\textit{e-sendsTo}) \sqsubseteq f'(\textit{e-sendsTo})$: Dies bedeutet, daß $C \notin f'(\textit{e-sendsTo})$. Somit erhalten wir $[\![e]\!]^{loc}(f) = [\![e]\!]^{loc}(f') = \bot_{fl}$, da die Ausgangsinformation nicht weitergeleitet wird.

Lemma 4.1.3

Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc}$, $e \in E^{ag}$, von MONO sind nicht distributiv.

Beweis:

Angenommen die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc}$, $e \in E^{ag}$, sind distributiv. Zu zeigen ist dann:

$$\forall \mathcal{FL}' \subseteq \mathcal{FL} : \; [\![e]\!]^{loc}(\bigsqcap \mathcal{FL}') = \bigsqcap \{[\![e]\!]^{loc}(f) \mid f \in \mathcal{FL}'\}.$$

Sei nun $f_1 \in \mathcal{FL}'$ mit $f_1(\textit{e-sendsTo}) = \{C\}$ und $f_2 \in \mathcal{FL}'$ mit $f_2(\textit{e-sendsTo}) = \{D\}$. Dann ist

$$\begin{aligned}
&[\![\textit{e-sendsTo } C.m_1 \colon \ldots m_n \colon]\!]^{loc}(\bigsqcap \{f_1, f_2\})(\textit{e-sendsTo}) \\
&= f_1 \sqcap f_2(\textit{e-sendsTo}) \\
&= \{C, D\}
\end{aligned}$$

und andererseits

$$\begin{aligned}
&[\![\ e\text{-}sendsTo\ C.m_1 \colon \ldots m_n \colon\ ]\!]^{loc}(f_1(e\text{-}sendsTo)) \\
&\sqcap\ [\![\ e\text{-}sendsTo\ C.m_1 \colon \ldots m_n \colon\ ]\!]^{loc}(f_2(e\text{-}sendsTo)) \\
&= f_1(e\text{-}sendsTo) \sqcap \bot_{fl} \\
&= \{C\}
\end{aligned}$$

und dies steht im Widerspruch zur Distributivität. Also sind die Filter-Funktionen nur monoton.

Mit Theorem 3.6.6, Lemma 4.1.1 und Lemma 4.1.2 erhalten wir sofort die Korrektheit der monotonen Typanalyse.

Theorem 4.1.4
Die *OMFP*-Lösung von MONO ist eine korrekte Approximation ihrer *OMOP*-Lösung.

4.1.3 Komplextität des Verfahrens

Im Gegensatz zu Bitvektor-Datenflußanalyseproblemen, die lineare Komplexität aufweisen, errechnet sich die Laufzeit-Komplexität im ungünstigsten Fall (worst-case) für die Typanalyse wie folgt:

Sei n die Größe des Programms, dann gilt:
$|\mathcal{C}'| \ = \ O(n), \ \ |\mathcal{V}'| \ = \ O(n),$
$|\mathcal{P}(\mathcal{C}')| \ = \ O(2^n)$, und
$|\mathcal{FL}| \ = \ |[\mathcal{V}' \to \mathcal{P}(\mathcal{C}')]| \ = \ O(2^{n^2}).$

Es werden Iterationen über einem Bereich von exponentieller Größe ausgeführt, um die Bedeutung $[\![\, e \,]\!]^{loc}$ eines Ausdrucks e zu berechnen. Das Verfahren hat also eine exponentielle Komplexität[2].

4.1.4 Grenzen der MONO-Typanalyse

Die soeben definierte Typanalyse erkennt viele Fehlerquellen in objekt-orientierten Programmen, aber allein schon die Tatsache, daß MONO nur eine monotone Typana-

[2]Die Verwendung von Verfahren mit exponentieller Komplexität ist für größere Programme in der Regel unrealistisch aufgrund der hohen Laufzeit. Wir werden später zeigen, daß eine Veränderung des generischen Algorithmus zwar an der theoretischen Komplexität nichts ändert, aber trotzdem bei den praktischen Anwendungen und Tests zu akzeptablen Ergebnissen führt.

lyse ist, deutet auf ihre Grenzen hin. Ein kleines Programmbeispiel soll zeigen, daß MONO gewisse Situationen nicht im Sinne des Laufzeitverhaltens behandeln kann. Genauer müßten wir hier von der *OMFP*-Lösung der Typanalyse MONO sprechen, denn nur diese kann praktisch berechnet werden. Betrachten wir das Laufzeitverhalten des folgenden Beispielprogrammes

```
class A
method m:x
  x k                                    (*6)
method k
  self
end A

class B
method m:y
  y l                                    (*5)
method l
  self
end B

class BspMono
var a b
method go
  a:=3;
  b:=4;
  if a=b
     then a:=A new; b:=A new             (*1)
     else a:=B new; b:=B new             (*2)
  endif;                                 (*3)
  a m:b                                  (*4)
end BspMono

BspMono new go
```

so ist nach Ausführung der Verzweigung (*3) die Variable a ein Objekt vom Typ B und die Variable b ebenso ein Objekt vom Typ B. Somit kommt es in (*4) zum Methodenaufruf m in der Klasse B mit Argumenttyp B. Dieser hat zur Folge, daß eine Methode l in der Klasse B aufgerufen wird (*5). Das Programm wird insgesamt korrekt ausgeführt. Da in diesem Programm keine rekursiven Aufrufe und keine Schleifen vorkommen, können wir uns die *OMOP*-Lösung von MONO ebenfalls anschauen. Es gibt zwei Programmpfade, die analysiert werden müssen. Zunächst

wird der `then`-Zweig analysiert. Danach sind die Variablen a und b vom Typ A. In (*4) wird also die Methode m in der Klasse A aufgerufen, und zieht einen Aufruf der Methode k in der Klasse A nach sich. Dieser Programmpfad wird insgesamt korrekt analysiert. Der zweite Programmpfad ist exakt derjenige, der auch zur Laufzeit ausgeführt wird, siehe oben. Die *OMOP*-Lösung von MONO würde ein solches Programm akzeptieren, aber leider ist diese Lösung i.a. nicht algorithmisch berechenbar.

Die *OMFP*-Lösung von MONO dahingegen liefert Fehler, die zur Laufzeit nicht eintreten können, siehe Abbildung 4.1. Genauer: Die Analyse der Verzweigungsbedingung kann nicht eindeutig ermitteln, welcher Zweig zur Laufzeit ausgeführt wird. Also müssen beide Zweige analysiert werden. Daher sind die Variablen a und b nach Beendigung der Verzweigung jeweils vom Typ $\{A, B\}$. Dies bedeutet für (*4) nun, daß die Methode m mit Argumenttyp $\{A, B\}$ sowohl in der Klasse A als auch in der Klasse B mit Argumenttyp $\{A, B\}$ analysiert werden muß. Die Analyse des Methodenaufrufs in der Klasse B bewirkt einen erneuten Methodenaufruf, diesmal der Methode l in den Klassen A und B. Dies ist aber nicht möglich, da in der Klasse A eine solche Methode nicht existiert. Die Typanalyse MONO akzeptiert ein derartiges Programm nicht.

4.2 Die Typanalyse MONO-ASS

Im Vergleich zu MONO wird in der Analyse MONO-ASS (Monotonic Data–flow Analysis–minus–Destructive–Assignments) nur die destruktive Behandlung von Zuweisungen verändert. Daher geben wir hier nur die neue Definition der lokalen semantischen Funktion für Zuweisungen an, alles andere bleibt unverändert:

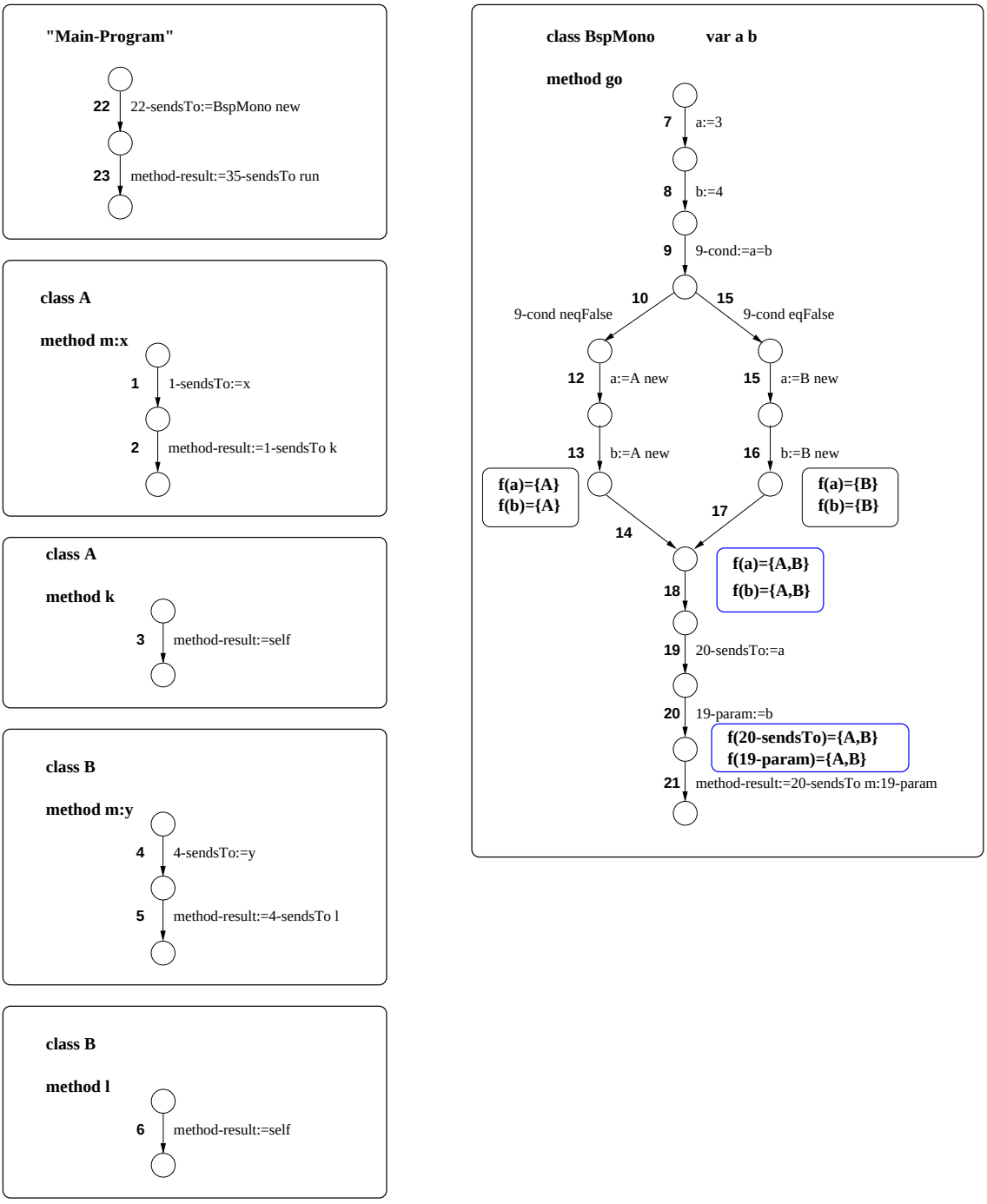


Abbildung 4.1: Von MONO nicht akzeptiertes Programm

- *Zuweisungen:* Wenn e mit einer Zuweisung der folgenden Form dekoriert ist, dann
  - $Id := SE$

$$[\![e]\!]^{loc}(f) =_{df} \begin{cases} f & \text{falls } f \in \{\bot_{fl}, \top_{fl}\} \\ f[Id \to \{A\} \cup f(Id)] & \text{falls } SE \equiv A\ \texttt{new} \\ f[Id \to \{Class(e)\} \cup f(Id)] & \text{falls } SE \equiv \texttt{self} \\ f[Id \to f(x) \cup f(Id)] & \text{falls } SE \equiv x,\ x \in \mathcal{V}' \\ f[Id \to \{Nil\} \cup f(Id)] & \text{falls } SE \equiv \texttt{nil} \\ f[Id \to \{Int\} \cup f(Id)] & \text{falls } SE \equiv Integer \end{cases}$$

Lemma 4.2.1
Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc},\ e \in E^{ag}$, von MONO-ASS sind monoton.

Theorem 4.2.2
Die *OMFP*-Lösung von MONO-ASS ist eine korrekte Approximation ihrer *OMOP*-Lösung.

Die Komplexität von MONO-ASS berechnet sich wie zuvor für MONO, und ist daher ebenfalls exponentiell.

4.2.1 Grenzen der MONO-ASS-Typanalyse

Betrachten wir folgendes Programm

```
class A
method m
  self
end A

class BspAss
var a
method go
  a:=1;
  a:=A new;          (*1)
  a m                (*2)
end BspAss

BspAss new go
```

so ist nach Ausführung der zweiten Zuweisung (*1) die Variable a vom Typ $\{Integer, A\}$. Somit kommt es in (*2) zu einem Methodenaufruf m in den Klassen $Integer$ und A. In der erstgenannten Klasse existiert aber eine derartige Methode nicht. Die Typanalyse MONO-ASS lehnt dieses Programm also ab, obwohl zur Laufzeit kein Probleme auftritt.

4.3 Die Typanalyse MONO-BRANCH

Die deterministische Behandlung von Verzweigungsausdrücken, sofern eindeutige Informationen vorliegen, wird in der Typanalyse MONO-BRANCH (Monotonic Data–flow Analysis-minus–Deterministic–Program-Branches) aufgehoben. Nur die beiden Filter, die Programmverzweigungen verwalten, werden derart modifiziert, daß nun beide Zweige eines *if-then-else* Ausdrucks analysiert werden:

- *Filter:* Wenn e mit einem Filter der folgenden Form dekoriert ist, dann
  - *e-cond* `eqFalse`
    $$[\![e]\!]^{loc}(f) =_{df} f$$
  - *e-cond* `neqFalse`
    $$[\![e]\!]^{loc}(f) =_{df} f$$

Lemma 4.3.1
Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc}$, $e \in E^{ag}$, von MONO-BRANCH sind monoton.

Theorem 4.3.2
Die *OMFP*-Lösung von MONO-BRANCH ist eine korrekte Approximation ihrer *OMOP*-Lösung.

Die Komplexität von MONO-BRANCH berechnet sich wie zuvor für MONO, und ist daher ebenfalls exponentiell.

4.3.1 Grenzen der MONO-BRANCH-Typanalyse

Die Analyse des Programms

```
class A
method m
  self
end A

class BspBranch
var a
method go
  if 1=1 then a:=A new
         else a:=1
  endif;                    (*1)
  a m                       (*2)
end BspBranch

BspBranch new go
```

ergibt, daß nach Auswertung (*1) der beiden Zweige des *if-then-else*-Ausdrucks die Variable a vom Typ $\{Integer, A\}$ ist. Somit tritt in (*2) ein Fehler auf, da die Methode m in der Klasse $Integer$ nicht bekannt ist. Die Typanalyse MONO-BRANCH lehnt ein solches zur Laufzeit durchaus korrektes Programm ab.

4.4 Die Typanalyse PS

Die originale monotone Typanalyse MONO wird nun für die beiden Eigenschaften "destruktive Zuweisungen" und "deterministische Verzweigungen" verändert. PS ist damit eine Zusammenfassung der beiden letzten Analysen. Der Name PS wurde anstelle von MONO-ASS-BRANCH (Monotonic Data–flow Analysis-minus-Deterministic-Branches-minus-Destructive-Assignments) gewählt, weil PS dem Typinferenzansatz aus [OPS92] gleicht. Die Änderungen bei den lokalen semantischen Funktionen sind nur bei Zuweisungen und den Filtern für Verzweigungen zu vollziehen, genau wie in den beiden vorherigen Abschnitten. Daher verzichten wir auf eine formale Ausführung.

Lemma 4.4.1
Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc},\ e \in E^{ag}$, von PS sind monoton.

Theorem 4.4.2
Die *OMFP*-Lösung von PS ist eine korrekte Approximation ihrer *OMOP*-Lösung.

Die Komplexität von PS berechnet sich wie zuvor für MONO, und ist daher ebenfalls exponentiell.

4.4.1 Grenzen der PS-Typanalyse

Da PS ein aus MONO-ASS und MONO-BRANCH zusammengesetzte Typanalyse ist, werden beide vorherigen Beispielprogramme in 4.2.1 und 4.3.1 von PS nicht akzeptiert.

4.5 Die Typanalyse DIST

In Abschnitt 4.1.4 haben wir ein Programm vorgestellt, daß von der monotonen Typanalyse MONO nicht akzeptiert wird, obwohl es zur Laufzeit korrekt ausführbar ist. Die Hauptursache liegt darin, daß nach Beendigung einer Verzweigung die über den linken Zweig und über den rechten Zweig berechneten Informationen vereint werden. Hierdurch kommt es zu einer Vermischung der eigentlich vorliegenden präzisen Typinformationen, die im weiteren nicht mehr rekonstruierbar sind. Die Art der Vereingung ist wesentlich für die Genauigkeit der gesamten Typanalyse. In diesem Abschnitt definieren wir eine präzise Typanalyse DIST (Distributive Data–flow Analysis), die über MONO hinausgehend

- an den Programmpunkten Datenflußinformationen nicht zusammenmengt, sondern die unterschiedlichen Informationen separat verwaltet mittels einer Menge von Datenflußinformationen; (dies geschieht durch den Übergang von einem Verband von Funktionen zu einem Potenzmengenverband von Funktionen);
- jeder Variablen genau eine Klasse zuordnet;
- implizit Methodenaufrufe polyvariant behandelt, d.h. ein Methodenaufruf wird für jede Kombination seiner Argumenttypen separat analysiert.

4.5.1 Spezifikation der Typanalyse

Sei im folgenden Π ein SOL -Programm und $T^{ag} = (N^{ag}, E^{ag}, s^{ag}, e^{ag})$ der zugehörige OO-Analyse-Graph. Dann definieren wir:

- $\mathcal{C}' =_{df} \mathcal{C}_{Id} \cup \{Int, False, True, Nil\}$ die Menge der Klassenidentifikatoren von Π erweitert um den Typ *Integer* als Repräsentant der Klasse der ganzen Zahlen und um die Typen *False*, *True* und *Nil* als Repräsentanten der speziellen Werte `false`, `true` und `nil`;
- $\mathcal{V}' =_{df} \mathcal{V}_{Id}(T^{ag})$ die Menge der in T^{ag} vorkommenden Variablenidentifikatoren und Argumentidentifikatoren;
- $\mathcal{P}$ bezeichne den Potenzmengenoperator.

Datenbereich und Datenflußinformationen

Der Datenbreich der Typanalyse wird definiert als der Verband der Menge der Funktionen

$$(\mathcal{L}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{P}(\mathcal{F}), \cup, \cap, \supseteq, \mathcal{F}, \emptyset)$$

mit

- $\mathcal{F} =_{df} [\mathcal{V}' \to \mathcal{C}'] \cup \{\top_f : \mathcal{V}' \to \{\Lambda\}\} \cup \{f_0 : \mathcal{V}' \to \{\}\}$,
- $\cup$, $\cap$, $\supseteq$, die die punktweisen Mengenoperationen Vereinigung, Schnitt und umfassend oder gleich sein darstellen,
- $\forall\, v \in \mathcal{V}' :\ \top_f(v) =_{df} \Lambda$ und $f_0(v) =_{df} \{\}$.

Die Funktion $\top_f$ gibt an, daß ein Methodenaufruf nicht ausgeführt ("verstanden") werden konnte. Programmpunkte, die nach Terminierung der Typanalyse mit $\emptyset$ annotiert sind, stellen unerreichbaren Programmcode dar und sind daher für das Programm überflüssig. Ein Programmpunkt, der mit $\mathcal{F}$ annotiert ist, deutet auf einen zur Laufzeit möglicherweise nicht ausführbaren Methodenaufruf hin.

Lokale abstrakte Semantik

Das Funktional

$$[\![\ ]\!]^{loc} : E^{ag} \to (\mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F}))$$

beschreibt die lokale abstrakte Semantik und wird definiert durch:

$$\forall\, e \in E^{ag}\ \forall\, F \in \mathcal{P}(\mathcal{F}).\ [\![\, e \,]\!]^{loc}(F) =_{df} \begin{cases} \mathcal{F} & \text{falls } \top_f \in F \\ \bigcup \{\, [\![\, e \,]\!]'(f) \mid f \in F \,\} & \text{sonst} \end{cases}$$

wobei $[\![\ ]\!]' : E^{ag} \to (\mathcal{F}\backslash\{\top_f\} \to \mathcal{P}(\mathcal{F}))$ definiert wird durch:

- *Zuweisungen:* Wenn e mit einer Zuweisung der folgenden Form dekoriert ist, dann
  - $Id := SE$

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f[Id \to A]\} & \text{falls } SE \equiv A\ \texttt{new} \\ \{f[Id \to Class(e)]\} & \text{falls } SE \equiv \texttt{self} \\ \{f[Id \to f(x)]\} & \text{falls } SE \equiv x,\ x \in \mathcal{V}' \\ \{f[Id \to Nil]\} & \text{falls } SE \equiv \texttt{nil} \\ \{f[Id \to Int]\} & \text{falls } SE \equiv Integer \end{cases}$$

  - $Id := Id_1\ \{+ | - | * | /\}\ Id_2$

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f[Id \to Int]\} & \text{falls } f(Id_1) = f(Id_2) = Int \\ \mathcal{F} & \text{sonst} \end{cases}$$

  - $e\text{-}cond := e\text{-}cond\ \texttt{instanceOf}\ C$

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f\} & \text{falls } f(e\text{-}cond) = C \\ \{f[e\text{-}cond \to False]\} & \text{sonst} \end{cases}$$

  - $e\text{-}cond := Id_1\ \{= | < | > | <= | >= | <>\}\ Id_2$:

$$[\![e]\!]'(f) =_{df} \{f[e\text{-}cond \to False], f[e\text{-}cond \to True]\}$$

- *Ausdrücke:* Wenn e dekoriert ist mit einem Ausdruck der Form SE, dann

$$[\![e]\!]'(f) =_{df} \{f\}$$

- *Filter:* Wenn e mit einem Filter der folgenden Form dekoriert ist, dann
  - $e\text{-}sendsTo\ C.m_1 : \ldots m_n :$

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f\} & \text{falls } f(e\text{-}sendsTo) = C \\ \emptyset & \text{sonst} \end{cases}$$

  - $e\text{-}sendsTo\ \texttt{SthOutOf}\ Impl(Method(e))$

$$[\![e]\!]'(f) =_{df} \begin{cases} \mathcal{F} & \text{falls } f(e\text{-}sendsTo) \notin Impl(Method(e)) \\ \emptyset & \text{sonst} \end{cases}$$

– *e-cond* `eqFalse`

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f\} & \text{falls } f(\textit{e-cond}) = \textit{False} \\ \emptyset & \text{sonst} \end{cases}$$

– *e-cond* `neqFalse`

$$[\![e]\!]'(f) =_{df} \begin{cases} \{f\} & \text{falls } f(\textit{e-cond}) \neq \textit{False} \\ \emptyset & \text{sonst} \end{cases}$$

Startinformation: ist die Menge $\{ f_0 \}$, bestehend aus der Funktion $f_0 \in \mathcal{F}$ mit $f_0(v) = \{\}$, $\forall\ v \in \mathcal{V}'$.

In der monotonen Typanalyse sind die Verwendung von Filtern, die destruktive Behandlung von Zuweisung und die möglichst deterministische Behandlung von Verzweigungen verantwortlich für die Mächtigkeit der Analyse. All diese Punkte sind in der distributiven Typanalyse ebenfalls involviert. Darüberhinaus ist die Trennung von Datenflußinformationen in Zuweisungen der Form *e-cond* := Id_1 {= | < | > | <= | >= | <>} Id_2, die Bedingungen darstellen, der entscheidende Punkt. Hierbei werden die zwei möglichen Datenflußinformationen getrennt voneinander gehalten. Die Aufgabe der Filter ist nun, die geeignete Typinformation aus der Menge der Datenflußinformationen zu selektieren, z.B. leitet der Filter *e-sendsTo* $C.m_1 : \ldots m_n :$ mit Eingabeinformationen $\{f_1, ..., f_k\}$ nur solche f_i weiter, bei denen der Empfänger *e-sendsTo* Instanz der Klasse C ist. Alle anderen Datenflußinformationen werden nicht weiter betrachtet und somit aus der Menge der Datenflußinformationen entfernt. Die gültigen f_i's werden dann unabhängig voneinander analysiert. Die Zusammenführung von Informationen z.B. nach Verzweigungen erfolgt als Vereingung der Datenflußinformationen, so daß eine Menge von Informationen entsteht. Dadurch wird an jedem Programmpunkt eine separate Behandlung der Typinformationen ermöglicht. Betrachten wir die Bearbeitung des BspMono-Programmes aus Abbildung 4.1, so bekommen wir nun exaktere Ergebnisse, siehe Abbildung 4.2.

4.5.2 Korrektheit der distributiven Typanalyse

Die beiden folgenden Lemmata bilden die Basis für die Koinzidenz der Typanalyse DIST.

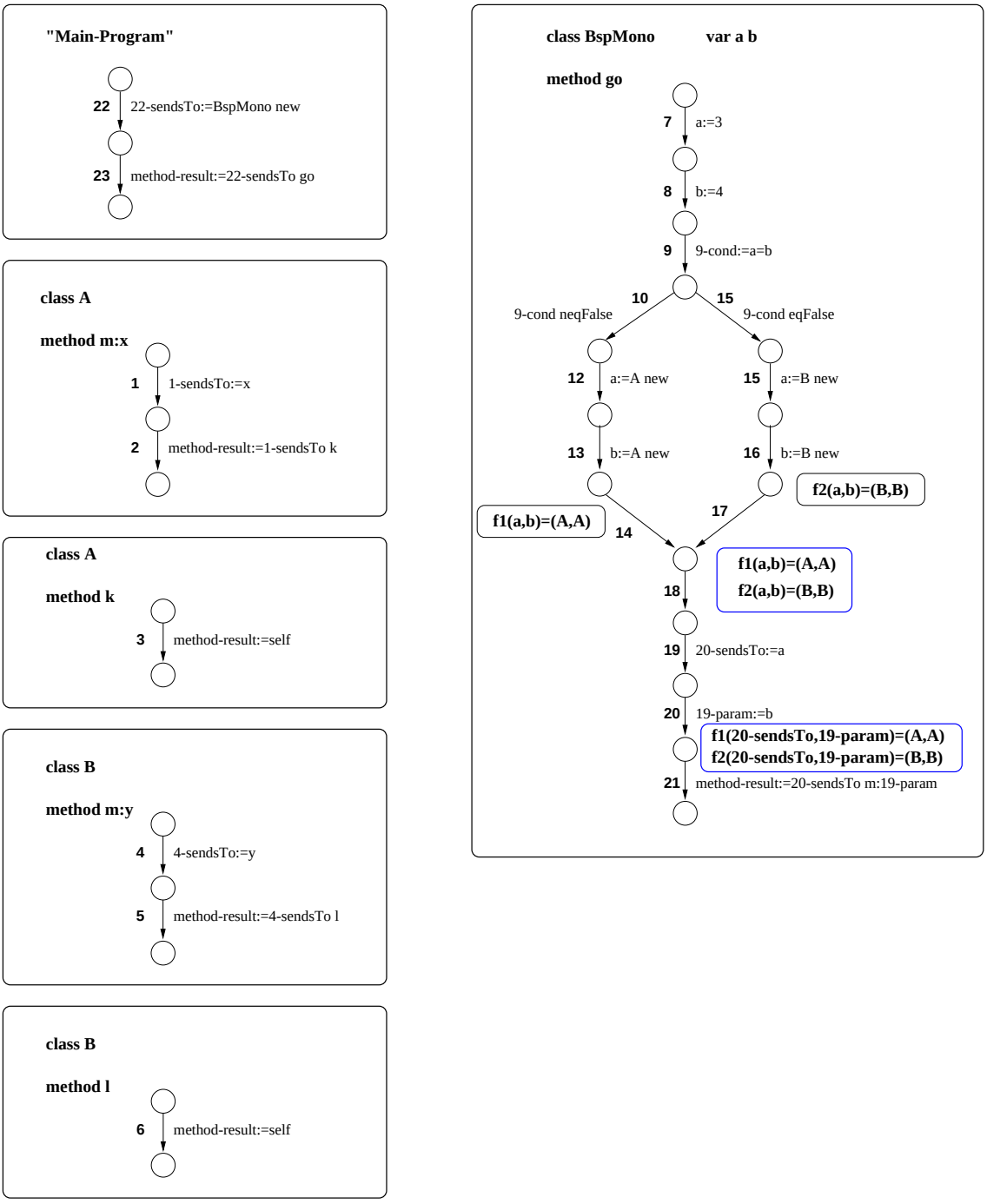


Abbildung 4.2: Von DIST akzeptiertes Programm

Lemma 4.5.1

$\mathcal{P}(\mathcal{F})$ ist ein vollständiger Verband, der die absteigende Kettenbedingung erfüllt.

Lemma 4.5.2

Die lokalen abstrakten Semantikfunktionen $[\![e]\!]^{loc},\ e \in E^{ag}$, von DIST sind distributiv.

Beweis:

Distributivität bedeutet hier:

$$\forall \mathcal{FM} \subseteq \mathcal{P}(\mathcal{F})\ [\![e]\!]^{loc}(\bigsqcap(\mathcal{FM})) = \bigsqcap\{[\![e]\!]^{loc}(F) \mid F \in \mathcal{FM}\}.$$

Da $\mathcal{F}$ und daher auch $\mathcal{P}(\mathcal{F})$ endlich sind und $\bigsqcap = \bigcup$, gilt:

$$\mathcal{FM} = \{F_1, ..., F_n\} \text{ mit } F_i = \{f_{i_1}, ..., f_{i_{m_i}}\} \text{ für } 1 \leq i \leq n.$$

$\forall F \in \mathcal{FM}$ gelte $\top_f \notin F$, da die Aussage des Theorems für diesen Fall trivialerweise erfüllt ist. Es gilt also:

$$\begin{aligned} \bigcup \mathcal{FM} &= F_1 \cup ... \cup F_n \\ &= \{f_{1_1}, ..., f_{1_{m_1}}, ..., f_{n_1}, ..., f_{n_{m_n}}\} \quad =: H \end{aligned}$$

und weiter

$$\begin{aligned} [\![e]\!]^{loc}(H) &= \bigcup\{[\![e]\!]'(f) \mid f \in H\} \\ &\quad [\![e]\!]'(f_{1_1}) \cup ... \cup [\![e]\!]'(f_{1_{m_1}}) \cup ... \cup [\![e]\!]'(f_{n_1}) \cup ... \cup [\![e]\!]'(f_{n_{m_n}}) \end{aligned}$$

Andererseits gilt ebenso:

$$\begin{aligned} &\bigcup\{[\![e]\!]^{loc}(F) \mid F \in \mathcal{FM}\} \\ &= [\![e]\!]^{loc}(F_1) \cup [\![e]\!]^{loc}(F_2) \cup ... \cup [\![e]\!]^{loc}(F_n) \\ &= [\![e]\!]'(f_{1_1}) \cup ... \cup [\![e]\!]'(f_{1_{m_1}}) \cup ... \cup [\![e]\!]'(f_{n_1}) \cup ... \cup [\![e]\!]'(f_{n_{m_n}}) \end{aligned}$$

Also sind alle lokalen semantischen Funktionen distributiv.

Mit Theorem 3.6.7 und Lemma 4.5.2 erhalten wir die gewünschte Koinzidenz der distributiven Typanalyse.

Theorem 4.5.3

Die *OMFP*-Lösung von DIST fällt (koinzidiert) mit ihrer *OMOP*-Lösung zusammen.

4.5.3 Komplextität des Verfahrens

Bei den monotonen Typanalysen ist die Komplexität schon exponentiell gewesen. Dies verändert sich bei der distributiven Analyse nochmals gewaltig. Die Komplexität von DIST berechnet sich wie folgt:
Sei n die Größe des Programms, dann

$|\mathcal{C}'| \;=\; O(n), \;\; |\mathcal{V}'| \;=\; O(n),$

$|\mathcal{F}| \;=\; |[\mathcal{V}' \to \mathcal{C}']| \;=\; O(n^n)$, und $\;\; |\mathcal{P}(\mathcal{F})| \;=\; O(2^{n^n})$.

Dies bedeutet, um die Bedeutung $[\![\, e \,]\!]^{loc}$ eines Ausdrucks e zu berechnen, wird über einen doppelt-exponentiell großen Bereich iteriert[3].

4.6 Vergleich der Mächtigkeiten der Typanalysen

Die Mächtigkeiten der fünf Typanalysen kann folgendermaßen zusammengefaßt werden:

- DIST akzeptiert mehr Programme als MONO,
- MONO akzeptiert mehr Programme als MONO-ASS,
- MONO akzeptiert mehr Programme als MONO-BRANCH,
- MONO-ASS akzeptiert mehr Programme als PS,
- MONO-BRANCH akzeptiert mehr Programme als PS.

[3]In den praktischen Tests wird sich diese Typanalyse als problematisch herausstellen. Die doppelt-exponentielle Komplexität scheint in der Praxis manchmal zu mächtig zu sein.

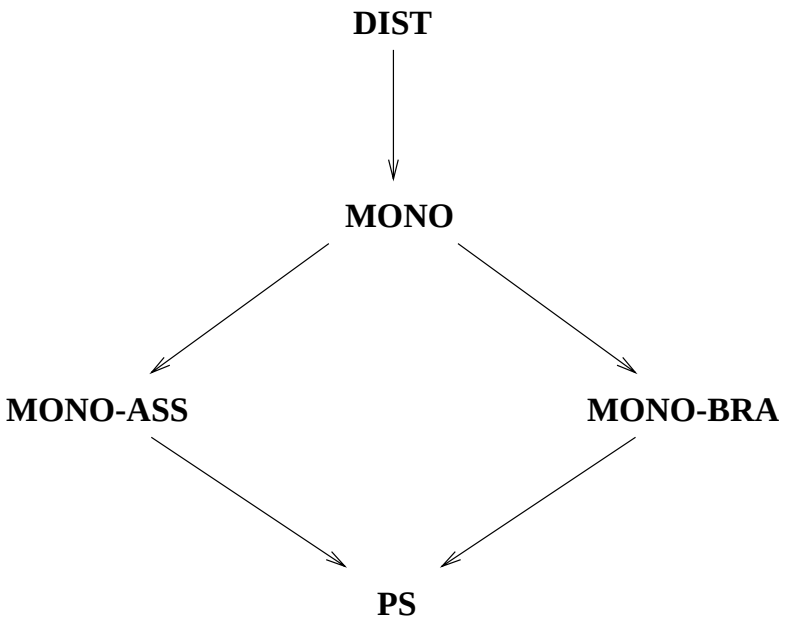


Abbildung 4.3: Hierarchie der Typanalysen

Kapitel 5

Typanalyse - Die Praktische Seite

Nachdem wir in den beiden vorangegangenen Kapiteln den theoretischen Rahmen für die Datenflußanalyse von objekt–orientierten Programmiersprachen sowie die Definition von verschiedenen Typanalysen präsentiert haben, wenden wir uns nun der Umsetzung und der Anwendung der Analysen zu.

5.1 Der Demand-Driven Algorithmus

In Kapitel 4 deuteten die Komplexitätsbetrachtungen der Typanalysen allerdings daraufhin, daß die Verwendung der Algorithmen 3.5.1 und 3.5.2 für "reale" Programme (mit mehr als 20 Variablen) hoffnungslos unrealistisch ist, da die Laufzeit exponentiell explodiert. Daher haben wir die beiden Algorithmen derart verändert, daß nur noch die für die Analyse und das Endergebnis relevanten Informationen berechnet werden. Die Komplexität der Typanalysen bleibt zwar unverändert exponentiell, aber wie wir später sehen werden, zeigten alle analysierten Programme ein gutartiges Laufzeitverhalten.

Die Algorithmen 3.5.1 und 3.5.2 werden zu einem Algorithmus verschmolzen, wobei eine *demand-driven* (*lazy*) Auswertungsstrategie verfolgt wird, sodaß nur diejenigen Teile der abstrakten Semantik von Methoden berechnet werden, die gerade benötigt werden.

Algorithmus 5.1.1 *Demand-driven Berechnung der OMFP-Lösung*
Eingabe:

- Ein OO-Analyse-Modell T^{am},
- ein vollständiger Verband $\mathcal{L}$, der die absteigende Kettenbedingung erfüllt,
- eine Startinformation $c_0 \in \mathcal{L}$,
- für jede Transition $e \in E^{am}$ die monotone lokale semantische Funktion $[\![e]\!]^{loc}$,
- für jede Transition $e \in E^{am}_{mc}$ und jede Klasse $C \in Impl(Method(e))$ die lokalen semantischen Funktionen $[\![e']\!]^{loc}$ für $e' \in \{e^C_{pb}, e^C_{fil}, e_{SthOutOf}\}$,
- und falls $e \in E^{am}_{assmc}$ auch die lokalen semantischen Funktion $[\![e^C_{ass}]\!]^{loc}$.

Ausgabe: Eine Annotation von T^{am} mit der *OMFP*-Lösung abgelegt in dfi.

Anmerkung: Die Variable *workset* steuert den iterativen Berechnungsprozeß. Die Elemente von *workset* sind Zustände in T^{ag}. Die Variable *meet* speichert das Ergebnis der jeweils letzten Berechnung von *meet*. Die Funktion $evalTr$ berechnet die notwendigen Transformationen demand-driven-artig. Die Funktion $ctr : MethodId \times \mathcal{L} \to \mathcal{L}$ (call transformation) berechnet die Approximation einer Methode bzgl. eines Zustandes, ähnlich wie *ltr*.

BEGINMAINPROGRAM
(Initialisierung der Annotierungs-Arrays gtr und ctr)
FORALL $n \in N$ **AND** $v \in \mathcal{L}$ **DO** $gtr[n, v] := \top$ **OD**;
FORALL $m \in \mathcal{M}_{Id}$ **AND** $v \in \mathcal{L}$ **DO** $ctr[m, v] := NOTEVAL$ **OD**;
(Initialisierung des Annotierungs-Arrays *dfi* und der Variablen *workset*)
FORALL $n \in N \backslash \{start(M_H)\}$ **DO** $dfi[n] := \top$ **OD**;
$dfi[start(M_H)] := c_0$;
$workset := \{ start(M_H) \}$;

(Iterative Fixpunkt-Berechnung)
WHILE $workset \neq \emptyset$ **DO**
  LET $m \in workset$
    BEGIN
      $workset := workset \backslash \{ m \}$;

(Neuberechnung der m spezifischen Informationen)

FORALL $e \in E$ with $source(e) = m$ **DO**

$meet := dfi[dest(e)] \sqcap evalTR(e, dfi[source(e)])$;

IF $dfi[dest(e)] \sqsupset meet$

THEN

$dfi[dest(e)] := meet$;

$workset := workset \cup \{ dest(e) \}$ **FI**;

IF $e \in E^{am}_{mc}$

THEN

FORALL $s \in \{start(Model(Method(e), C)) \mid C \in Impl(Method(e)) \}$

DO

$meet := dfi[s] \sqcap [\![e^{Class(s)}_{pt}]\!]^* \circ [\![e^{Class(s)}_{fil}]\!]^*(dfi[source(e)])$;

IF $dfi[s] \sqsupset meet$

THEN

$dfi[s] := meet$;

$workset := workset \cup \{ s \}$ **FI**

OD

FI

OD

END

OD

ENDMAINPROGRAM.

FUNCTION $evalTR(e' : Transition;\ \hat{v} : \mathcal{L}) : \mathcal{L}$

IF $e' \in E \backslash E_{call}$ **THEN RETURN** $[\![e']\!]^*(\hat{v})$ **FI**;

IF $ctr[Method(e'), \hat{v}] \neq NOTEVAL$ **THEN**

IF $e' \in E^{am}_{assmc}$ **THEN RETURN** $[\![e'^C_{ass}]\!]^*(ctr[Method(e'), \hat{v}])$

ELSE RETURN $ctr[Method(e'), \hat{v}]$ **FI**

FI

(Initialisierung der Variablen $workset$)

$workset := \{ (s, v', \hat{v}) \mid s \in start(Model(Method(e'), C)),\ v' = [\![e'^C_{pt}]\!]^* \circ [\![e'^C_{fil}]\!]^*(\hat{v}),$

$C \in Impl(Method(e')) \}$;

$ctr[Method(e'), \hat{v}] := \top$;

(Iterative Fixpunkt-Berechnung)

WHILE $workset \neq \emptyset$ **DO**

LET $(m, f, v) \in workset$

BEGIN

$workset := workset \backslash \{ (m, f, v) \};\ meet := gtr[m, v] \sqcap f$

IF $gtr[m, v] \sqsupset meet$

THEN

$gtr[m, v] := meet;$

IF $m \in \{end(\mathcal{M}_{Mdl})\}$

THEN

FORALL $e \in Sender(mm(m))$ **AND FORALL** $\tilde{v} \in \mathcal{L}$

WITH $gtr[source(e), \tilde{v}] = v$ **AND** $\tilde{v} \neq \top$ **DO**

$ctr[Method(e), v] :=$

$$\bigsqcap \{ gtr[end(Model(Method(e), C)), v] \mid C \in Impl(Method(e)) \};$$

$$workset := (workset \cup \{ (dest(e), ctr[Method(e), v], \tilde{v}) \}) \backslash \{ (dest(e), f', \tilde{v}) \in workset \mid f' \sqsupset ctr[Method(e), v] \}$$

OD

ELSE

$$workset := (workset \cup \{ (dest(e), evalTR(e, gtr[m, v]), v) \mid e \in E \text{ with } source(e) = m \}) \backslash \{ (dest(e), f', v) \in workset \mid f' \sqsupset evalTR(e, gtr[m, v]) \}$$

FI

FI

END

OD

IF $e' \in E^{am}_{assmc}$ **THEN RETURN** $[\![e'^{C}_{ass}]\!]^*(ctr[Method(e'), \hat{v}])$

ELSE RETURN $ctr[Method(e'), \hat{v}]$ **FI**.

Die wesentlichen Veränderungen der demand-driven Variante gegenüber der ursprünglichen Algorithmen 3.5.1 und 3.5.2 lassen sich durch folgende Punkte beschreiben:

1. Das Semantik-Funktional $[\![\ ]\!]$ in Alg.3.5.1+3.5.2, daß durch den Präprozeß im Alg.3.5.2 bereits vollständig berechnet worden ist, wird in der demand-driven Version ersetzt durch den Aufruf von $evalTr$ mit einer Variablenbelegung v. Die Funktion $evalTr$ berechnet die Transformation einer Transition. Diese kann einen einfachen Ausdruck darstellen, wobei dann sofort die geeignete Transformation $[\![\, e \,]\!]^{loc}(v)$ ausgeführt werden kann. Falls aber ein Methodenaufruf durch die Transition dargestellt wird, so überprüft $evalTr$ zunächst, ob ein solcher Methodenaufruf bezüglich der Variablenbelegung v bereits berechnet wurde. Wenn dies der Fall ist, so kann der geeignete Wert sofort zurückgegeben werden. Ansonsten wird der gesuchte Wert berechnet.

2. Die lokale Transformation *ltr* in Alg.3.5.1+3.5.2 wird in Alg.5.1.1 ersetzt durch ein Aufruf-Transitions-Feld *ctr* (call transformation) mit $ctr : \mathcal{M}_{Id} \times \mathcal{L} \to \mathcal{L}$. Der Grund hierfür ist darin zu sehen, daß eine transitionsunabhängige Repräsentation für folgende Situationen gefunden werden mußte:

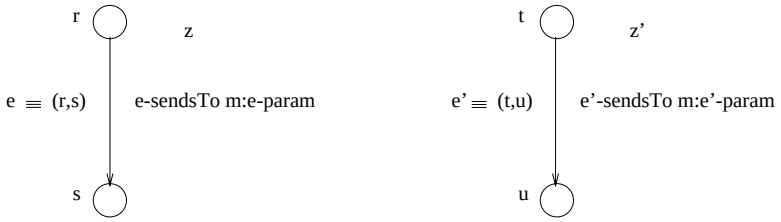


Abbildung 5.1: Überdeckung von Transitionen

Wenn die Datenflußinformationen z und z', *e-sendsTo* und *e'-sendsTo* sowie *e-param* und *e'-param* zusammenfallen, so mußte trotzdem in Alg.3.5.1+3.5.2 die *ltr*-Transformation für beide Transitionen berechnet werden, da *ltr* abhängig von der jeweiligen Transition ist. Die *ctr*-Transformation dahingegen ist von dem Methodenidentifikator abhängig, der in der Transition vorkommt. Somit

haben wir für $z = z'$ eine Überdeckung der beiden Transitionen bekommen.[1] Die Transition (t, u) muß nicht neu analysiert werden, da sie mit der Transition (r, s) zusammenfällt. Die *ctr*-Transformation berechnet nur die Aufruf-Transformation, nicht die Wertrückgabe, falls der Aufruf in einer Zuweisung vorkommt. Die Wertrückgabe wird von $evalTr$ erst am Ende in Alg.5.1.1 berechnet.

3. Die Elemente der *workset* in $evalTr$ sind um eine Komponente erweitert worden. In Alg.3.5.1+3.5.2 stellt die zweite Komponente der *workset* eine Funktion der Form $f : \mathcal{L} \to \mathcal{L}$ dar. In Alg.5.1.1 wird die *workset* initialisert mit (a) dem Startzustand der Methode, (b) der Information v' nach Auswertung der Filter-Transition und den Parameterbindungen angewandt auf die Information $\hat{v}$, die an der Aufruf-Transition vorlag und (c) eben dieser Information $\hat{v}$ als dritte Komponente. Warum diese Information benötigt wird, werden wir gleich sehen. Nach der Initialisierung der *workset* wird die Aufruf-Transformation *ctr* bezüglich des Methodennamens und der Information $\hat{v}$ mit $\top$ initialisiert.

4. Die globale Transformation *gtr* in Alg.5.1.1 merkt sich in ihrer zweiten Komponente die Startinformation der Aufruf-Transition. Der Schnitt(meet)-Operator wird weiterhin mit der zweiten Komponente der *workset* ausgeführt. Da die Filter-Transitionen und die Parameterbindungen bereits bei der Initialisierung der *workset* angewandt worden sind, werden sie bei der Wertzuweisung an *ctr* nicht mehr benötigt und daher weggelassen.

5. Ein sehr wichtiges demand-driven Detail ist bei der Neuauswertung aller ähnlichen Methodenaufrufe zu finden. An dieser Stelle schlägt in Alg.3.5.1+3.5.2 die Laufzeitkomplexität zu: es werden alle Methodenaufrufe gleichen Namens mit **allen** darstellbaren Informationen neu berechnet. Aber genau dies ist viel zuviel. Veränderungen kann es nur bei ähnlichen Aufruf-Transitionen mit gleichem Startzustand geben, siehe Abbildung 5.1. Also brauchen auch nur diese

[1] In der Implementierung von SOLAT werden die eindeutigen Variablenidentifikatoren des Empfängers und der Argumente eines Methodenaufrufes in globale feststehende Variablenidentifikatoren umbenannt. D.h. *e-sendsTo* und *e'-sendsTo* werden umbenannt in *sends-To* und *e-param* wird umbenannt in *Param-1*, siehe auch [Po97]. Somit enthalten sowohl z als auch z' die geeigneten Informationen, um den Vergleich von z und z' durchzuführen.

Transitionen neu analysiert zu werden. Die bereits berechneten Zustände sind in *gtr* abgelegt. Es muß für den Startzustand einer ähnlichen Aufruf-Transition überprüft werden, ob es eine Berechnung von *gtr* bereits gegeben hat, die an der Aufruf-Transition die Information v liefert.

6. Die *workset* wird an zwei Stellen vergrößert. In Alg.5.1.1 geschieht dies differenzierter als in Alg.3.5.1+3.5.2, indem jeweils überflüssige Transitionen und Informationen nicht weiter bearbeitet werden. Wenn ein neues Element in die *workset* aufgenommen wird, so werden gleichzeitig alle Elemente mit gleichen Zuständen und ungenaueren Informationen aus der *workset* entfernt.

Dieser demand-driven Algorithmus ist in GNU-C++ implementiert worden, siehe [Po97] und daher für verschiedene Hardware-Plattformen verwendbar.

5.2 Testprogramme

Unsere Suite von Testprogrammen umfaßt 9 SOL -Programme unterschiedlicher Größenordnung. Sie varieren zwischen einem kleinen Programm mit 30 Zeilen (*lines of code*), sechs mittelgroßen Programmen mit 106-295 Zeilen und zwei "großen" Programmen mit 1005 bzw. 1652 Zeilen. Im einzelnen sind dies:

- *Peano* ist ein Programm, das die Peano Integerzahlen implementiert, ursprünglich entwickelt worden von Johnson in Typed Smalltalk [GJ90];
- *Trees* and *Container* sind Adaptionen von Programmen, die von Palsberg und Schwartzbach [OPS92] als Testprogramme, ebenso wie *Peano*, verwendet wurden;
- *N-Queen*, *Quicksort*, *Tower of Hanoi* und *Sieve* sind Programme, die bekannte Problemstellungen lösen;
- *Diff* ist ein Programm, daß das Unix-Kommando diff implementiert, auch als Wolczko's Version des diff Kommandos bekannt [HS77, HU94];
- *Richards* beinhaltet einen Betriebssystem-Simulator. *Diff* und *Richards* sind die beiden großen Programme, die auch als Benchmark-Programme von der

Self-Gruppe immer verwendet werden [Ch92, HCU91, HU94, Ag94, Ag95a, AH95].

Die folgenden Testergebnisse wurden auf einer DEC Alpha Station 600 5/266 mit 256MB RAM erzielt.

5.3 Laufzeiten der Typanalysen

Laufzeiteffizienz ist ein ultimatives Ziel einer Implementierung. Laufzeituntersuchungen sind besonders wichtig, wenn die zugrundeliegenden Algorithmen eine theoretische exponentielle bzw. doppelt-exponentielle Komplexität aufweisen. Wir betrachten daher zuächst die erzielten Resultate in Abbildung 5.2.

Name	loc	var	m.defs	MONO	M.A.	M.B.	PS	DIST
Peano	128	6	42	0.57	0.48	22.20	16.90	0.08
Trees	186	10	40	0.08	0.46	2.23	2.23	0.08
Container	30	3	5	0.02	rej.	0.02	rej.	0.02
N-Queen	106	8	18	0.40	0.40	0.50	0.49	0.05
Quicksort	118	14	12	0.25	0.25	0.24	0.25	0.26
Tower of Hanoi	295	24	36	4.60	4.58	4.55	4.57	476.12
Sieve	107	12	11	0.19	0.19	0.19	0.20	0.25
Diff	1652	121	218	819.90	rej.	rej.	rej.	OutOfMem
Richards	1005	179	122	7.33	rej.	rej.	rej.	7.18

Abbildung 5.2: Analysezeiten der fünf Typanalysen in sec.: loc=Anzahl der Programmzeilen(lines of code), var=Anzahl der Variablen, m.defs=Anzahl der Methodendefinitionen.

Die Laufzeiten der MONO-Analyse sind mit Ausnahme des *Diff*-Programms sehr gut. Dies gilt z.T. auch für die anderen Typanalysen, wobei die monotonen Analysen allerdings zwei bzw. drei Programme, darunter jeweils die beiden großen Programme, nicht erfolgreich verarbeiten konnten. Die distributive Analyse zeigte teilweise sehr vielversprechende Resultate, benötigt aber bei dem größten Programm *Diff* soviel

Speicher (>800 MB), daß die Analyse auf den zur Verfügung stehenden Rechnern nicht normal beendet wurden.

Zunächst könnte man vermuten, daß die Laufzeiten in direktem Zusammenhang mit den Programmgrößen oder der Größe der Datenflußinformationen (Anzahl der Variablen) stehen. Dies konnte aber durch die Messungen nicht bestätigt werden. Wenn wir die Ergebnisse der MONO-Analyse betrachten, so erhalten wir folgende Situation. Das *Trees*-Programm (186 lines of code) benötigt nur 0.08 sec, wohingegen *Peano* (128 loc) mit 0.57 sec 7 mal langsamer analysiert wird. Die Programme *N-Queen* und *Sieve* sind von vergleichbarer Größe, aber die Analyse von *N-Queen* benötigt doppelt soviel Zeit wie die Analyse von *Sieve*. Wenn wir die Anzahl der Variablen betrachten, so sind *Richards* und *Diff* zwei Programme mit vielen Variablen, haben aber sehr unterschiedliche Laufzeiten. *Richards* ist um den Faktor 115 schneller als *Diff*. Das *N-Queen*-Programm hat nur 8 Variablen, wird aber fünfmal langsamer analysiert als das *Trees*-Programm mit vergleichbarer Variablenanzahl. Es besteht also kein direkter Zusammenhang zwischen der Anzahl der Variablen und der Laufzeit der Analysen.

Die Ergebnisse von MONO-ASS zeigen, daß die Analysen von 40% aller Programme schlechter sind als im Vergleich zu MONO. Dies liegt vor allem darin begründet, daß drei Programme abgelehnt (rejected) werden und das *Trees*-Programm eine wesentlich schlechtere Analysezeit hat als bei MONO (Faktor 6).

Bei der Analyse mit MONO-BRANCH können wir erkennen, daß sie im Vergleich zu MONO nur vier Programme in ähnlicher Zeit bearbeitet. Zwei Programme werden von MONO-BRANCH abgelehnt und drei Programme enden mit schlechterer Laufzeit als dies bei MONO der Fall ist.

Der Vergleich von MONO-ASS und MONO-BRANCH bzw. PS liefert, daß MONO-ASS zwei Programme schneller verarbeitet als die anderen beiden monotonen Typanalysen und daß MONO-BRANCH ein Programm (*Container*) akzeptiert, das von den anderen beiden abgelehnt wird.

Die Ergebnisse der monotonen Analysen zeigen aber auch, daß eine unschärfere Typanalyse durchaus schneller erfolgreich ausgeführt werden kann als eine schärfere Typanalyse. Dieser Fall tritt bei den verschiedenen Typanalysen von *Peano* auf. Die Typanalyse MONO benötigt ca 20% mehr Analysezeit als die abgeschwächte Typ-

analyse MONO-ASS und die Typanalyse PS ist ebenfalls um ca 20% schneller als die Typanalyse MONO-BRANCH. Erklärbar ist dies dadurch, daß in einem Analyseschritt einer Typanalyse Informationen berechnet werden, die bei einer präziseren Analyse erst nach mehreren Berechnungsschritten entstehen.

Einen zwiespältigen Eindruck hinterläßt der Vergleich der Ergebnisse von der aggressivsten Typanalyse, DIST, und der monotonen Analyse MONO. Auf der einen Seite zeigt DIST bei drei Programmen (*Peano*, *N-Queen* und *Richards*) ein wesentlich besseres Verhalten als MONO. Umgekehrt analysiert MONO die beiden Programme *Tower of Hanoi* und *Sieve* bedeutend schneller. Aber ernsthaft problematisch ist das Verhalten der DIST-Analyse im Falle der Programme *Tower of Hanoi* und *Diff*. Bei dem *Tower of Hanoi*-Programm ist die Analysezeit 100mal langsamer. Wie bereits zuvor erwähnt, kann die Analyse des *Diff*-Programms nicht ausgewertet werden, da der große Speicherbedarf, über 800 MB, zu einem frühzeitigen Abbruch führt.

Aus dem Gesichtspunkt der Analysezeiten hat MONO das stabilste Verhalten gezeigt. DIST kann in manchen Fällen zu Verbesserungen führen (Faktor 7), kann aber auch das Gegenteil bewirken (Faktor 100). MONO-ASS, MONO-BRANCH und PS scheitern bei großen Programmen. Ansonsten hat MONO-ASS Vorteile gegenüber MONO-BRANCH und PS. Die beiden letztgenannten haben fast identische Laufzeiten.

Der Vollständigkeit halber ist auch die Anzahl der analysierten Transitionen (Transitionen, die in der *workset* bearbeitet werden) protokolliert worden. Die Ergebnisse haben wir in Tabelle 5.3 zusammengefaßt.

5.4 Qualität der Typanalysen

Insgesamt fünf Typanalysen mit unterschiedlicher Aggressivität (Präzision) wurden untersucht. Die Qualität einer Typanalyse kann anhand der Anzahl der nicht eindeutigen Methodenaufrufe gemessen werden. Ein Methodenaufruf wird eindeutig von der Typanalyse bestimmt, falls die Typanalyse herausfindet, daß der Empfängertyp des Aufrufs aus genau einem Element besteht. Ein Methodenaufruf ist mehrdeutig (oder nicht eindeutig), wenn der Empfängertyp des Aufrufs aus mehreren Elementen besteht. Verschiedene experimentelle Untersuchungen, vor allem die sehr differen-

Name	loc	var	m.defs	MONO	M.A.	M.B.	PS	DIST
Peano	128	6	42	1249	1108	9113	7525	270
Trees	186	10	40	55	972	2638	2638	55
Container	30	3	5	67	rej.	67	rej.	67
N-Queen	106	8	18	1329	1329	1401	1401	251
Quicksort	118	14	12	852	852	852	852	1063
Tower of Hanoi	295	24	36	4387	4387	4387	4387	17537
Sieve	107	12	11	715	715	747	747	826
Diff	1652	121	218	43522	rej.	rej.	rej.	OutOfMem
Richards	1005	179	122	4234	rej.	rej.	rej.	4859

Abbildung 5.3: Anzahl der bearbeiteten Transitionen.

zierten Untersuchungen der Self-Gruppe [Ch92, HU94, Hö95, Ag95b] haben gezeigt, daß in objekt–orientierten Programmen relativ wenige mehrdeutige Methodenaufrufe vorkommen. In Tabelle 5.4 haben wir die Anzahl der mehrdeutigen Methodenaufrufe, die die jeweiligen Typanalysen herausfinden, zusammengestellt.

Die Typanalyse MONO zeigt ein sehr gutes Verhalten, indem 6 der 9 Programme keine mehrdeutigen Methodenaufrufe enthalten. Die Analyse der drei "mehrdeutigen" Programme, hierzu gehört nur eines der beiden großen Programme, findet heraus, daß nur 0.47% (bei *Diff*), 12% (bei *Peano*) und 29% (bei *N-Queen*) aller Methodenaufrufe, die im Programm vorkommen, mehrdeutig sind.

MONO-ASS erkennt genau soviele eindeutige Methodenaufrufe wie MONO, mit Ausnahme der drei Programme, die MONO-ASS als fehlerhaft bezeichnet.

MONO-BRANCH und PS liefern fast identische Ergebnisse und führen im Vergleich zu MONO bei nur 5 bzw. 4 Programmen zu gleichen Ergebnissen.

Wie nach den Laufzeitergebnissen zu erwarten war, ergibt sich bei der distributiven Typanalyse wieder ein uneinheitliches Bild. Abgesehen von dem durch DIST nicht analysierbaren Programm, kann die distributive Analyse zwar die Ungenauigkeiten weiter verringern, bei *Peano* um fast die Hälfte und bei *N-Queen* um den Faktor 6, aber leider reicht eben nicht immer der Hauptspeicher aus.

Alle Typanalysen finden als Nebenprodukt zur (dynamischen) Laufzeit nicht

Name	Call Sites	MONO	M.A.	M.B.	PS	DIST
Peano	43	5	5	24	24	3
Trees	87	0	0	0	0	0
Container	7	0	rej.	0	rej.	0
N-Queen	21	6	6	9	9	1
Quicksort	19	0	0	0	0	0
Tower of Hanoi	67	0	0	0	0	0
Sieve	14	0	0	0	0	0
Diff	423	2	rej.	rej.	rej.	OutOfMem
Richards	226	0	rej.	rej.	rej.	0

Abbildung 5.4: Mehrdeutige Aufrufstellen von Methoden (not unique call sites, NUCS) in den Programmen.

erreichbare Programmteile heraus, sogenannter *dead code*, siehe Tabelle 5.5. Diese Programmteile werden von der Typanalyse mit $\top$ gekennzeichnet und brauchen nicht übersetzt zu werden, sondern können problemlos eliminiert werden, wodurch das ursprüngliche Programm verkleinert wird. Das Erkennen von dead code ist ein weiteres Qualitätsmerkmal einer Typanalyse.

Die MONO-Analyse findet im wesentlichen 44-85% der Methodendefinitionen als überflüssigen Code heraus, bei MONO-ASS sind dies 44-55% und bei MONO-BRANCH bzw. PS 33-50% bzw. 31-50%. Die präzise DIST-Analyse entdeckt 44-90% des ursprünglichen Codes als dead code. Beim Vergleich von DIST und MONO hat DIST bei vier Programme Vorteile gegenüber MONO (7-16% mehr erkannter dead code).

5.5 Speicherplatzbedarf

Die Typanalyse DIST sprengt den zur Verfügung stehenden Speicher bei dem *Diff*-Programm. Dadurch entstand die Idee, den Speicherplatzbedarf der Typanalysen genauer zu untersuchen, siehe Tabelle 5.6[2].

[2]Der angegebene Speicherplatzbedarf von weniger als 3.5 MB sind Durchschnittswerte, da bei wiederholten Messungen starke Schwankungen aufgetreten sind; einerseits wurden 1.5 MB und andererseits auch 3.5 MB für das gleiche Programm gemessen.

Name	m.defs	Mono	M.A.	M.B.	PS	Dist
Peano	42	21	21	6	6	24
Trees	40	34	22	14	14	36
Container	5	0	rej.	0	rej.	0
N-Queen	18	1	1	0	0	4
Quicksort	12	6	6	6	6	6
Tower of Hanoi	36	16	16	16	16	16
Sieve	11	5	5	5	5	5
Diff	218	131	rej.	rej.	rej.	OutOfMem
Richards	122	61	rej.	rej.	rej.	86

Abbildung 5.5: Nicht benötigte Methodendefinitionen (dead code).

Name	Mono	M.A.	M.B.	PS	Dist
Peano	2.5	2.5	4.7	4.5	2.5
Trees	2.5	3.5	4.5	4.5	2.5
Container	n.m.	rej.	n.m.	rej.	n.m.
N-Queen	2.5	3.7	3.7	3.7	2.5
Quicksort	3.7	3.7	3.7	3.7	4
Tower of Hanoi	5.9	5.9	5.9	5.9	161.0
Sieve	3.7	3.7	3.7	3.7	3.9
Diff	57.7	rej.	rej.	rej.	OutOfMem
Richards	8.2	rej.	rej.	rej.	17.2

Abbildung 5.6: Speicherplatzbedarf in MB; n.m.=nicht messbar.

Wiederum liefert die monotone Typanalyse MONO die besten Resultate. Bei fast allen Programmen liegt der Platzbedarf unter 6 MB. Nur im Falle des *Diff*-Programmes schnellt der Speicherbedarf auf 56 MB hoch. Die distributive Analyse liegt ebenfalls in diesem Rahmen, abgesehen von ihrem "Problem"-Programm und dem *Tower of Hanoi*-Programm. Beim *Richards*-Programm ist der Platzbedarf mit 17 MB im Verhältnis zur Programmgröße recht moderat. Die Analyse des *Tower of Hanoi*-Programms benötigt 161 MB, wobei die übrigen Ergebnisse für NUCS und dead code im Vergleich zur MONO-Analyse identisch sind. Bei den anderen monotonen Typanalysen liegt der Speicherbedarf unterhalb von 6 MB. Somit bewegt sich der Speicherplatzbedarf der monotonen Typanalysen in einem durchaus realistischem Bereich[3].

5.6 Bewertung der Ergebnisse

Betrachtet man alle Gesichtspunkte (Analysezeit, Qualität der Analyse, Speicherbedarf) zusammen, so gibt es einen eindeutigen Gewinner unter den Typanalysen: die monotone Typanalyse MONO. Sie zeigt gute bis sehr gute Analysezeiten mit recht hoher Präzision und moderatem Speicherbedarf.

Aber auch ein kombiniertes Vorgehen ist denkbar, um die Vorteile anderer Analysen mit einzubauen, wobei sich zwei Alternativen anbieten:

- Liefert MONO ein nicht zufriedenstellendes Ergebnis, so kann das zu analysierende Programm mit DIST erneut analysiert werden.
- Ist die Laufzeit von DIST zu lang bzw. benötigt DIST zuviel Speicher, so kann das zu analysierende Programm erneut mit MONO analysiert werden.

Die Testergebnisse haben auch die Wichtigkeit der Verwendung von destruktiven Zuweisungen und der möglichst deterministischen Behandlung von Programmverzweigungen in Typanalysen unterstrichen. Dabei ist vor allem die deterministische Behandlung von Programmverzweigungen hervorzuheben. Die Bedeutung von separater Behandlung von Methodenaufrufen muss nicht extra erwähnt werden.

[3]Vergleiche mit anderen Typanalysen konnten nicht angestellt werden, da hierfür keine Resultate publiziert worden sind, zumindestens dem Autor nicht bekannt sind.

5.7 Behandlung von uninitialisierten Variablen

Die Startinformationen der Typanalysen sind derart definiert, daß jeder Programmvariablen die leere Menge zugeordnet wird, d.h. zu Beginn der Analyse liegen keine Typinformationen vor. Betrachten wir die dynamische Semantik von SOL , so beinhaltet diese Wahl der Startinformation bereits eine gewisse Unschärfe. Eine höhere Präzision würden wir durch die Startinformation $f_0(v) = \{Nil\}$ erhalten[4]. Aber damit handelt man sich ein anderes Problem ein. Zunächst illustrieren wir den vorliegenden Sachverhalt an dem Beispiel in Abbildung 5.7. Die Variable a wird vor der

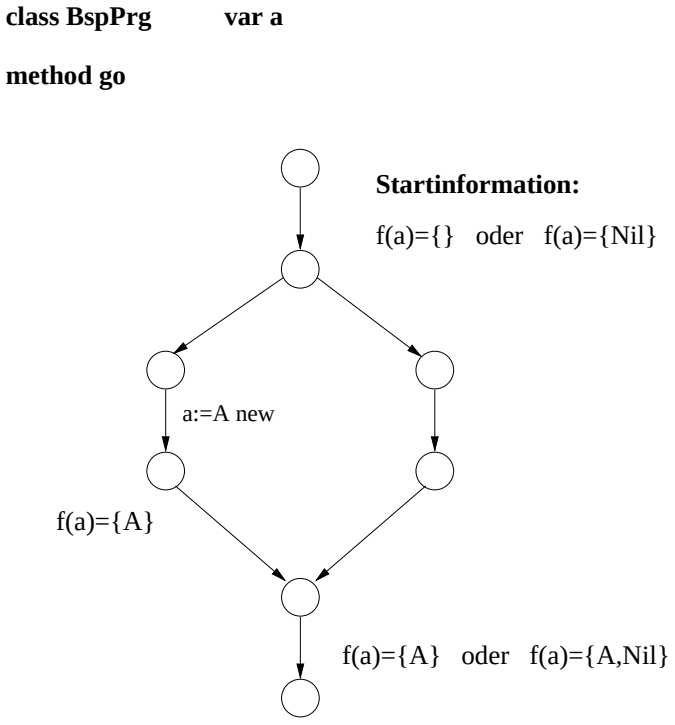


Abbildung 5.7: Problem der Startinformation

Programmverzweigung nicht explizit initialisiert. Angenommen beide Zweige müssen analysiert werden, so erhalten wir nach der Zusammenführung der beiden Zweige,

[4]Eine kleine Bemerkung am Rande: Die verwendete Startinformation $f_0(v) = \{\}$ war ursprünglich nicht geplant gewesen, sondern wurde durch einen Fehler in der Implementierung erzielt. Nachdem dieser Fehler entdeckt und korrigiert worden ist, trat aber ein gewaltiges Problem auf. Fast alle Programme wurden von den Typanalysen abgelehnt. Die Typanalysen MONO-ASS und PS waren unsinng geworden, da alle Variablen an jedem Programmpunkt mindestens mit Nil belegt waren.

daß die Variable a vom Typ $\{A\}$ bzw. vom Typ $\{A, Nil\}$ ist. Letzteres wird durch die exaktere Startinformation erzielt.

Fast alle Testprogramme beinhalten Abfragen, ob eine gewünschte Programmsituation vorliegt oder nicht (z.B. Überpüfung auf Indexgrenzen: ist der momentan vorliegende Index im definierten Indexbereich; wenn nein, dann wird diese Fehlersituation vom Programm (Programmierer) abgefangen). Die Analyse geht aber im weiteren davon aus, daß beide Fälle (Situation liegt vor oder liegt nicht vor) eintreten können. Dies führt dann während der Typanalyse zu häufig nicht gewünschten Effekten: es werden Situationen analysiert, die zur Laufzeit nicht vorkommen können. Somit haben wir eine "Überspezifikation" der Typanalyse erreicht. Unsere Tests und detaillierten Untersuchungen der Testprogramme haben dieses bestätigt. Fast alle Testprogramme wurden von den Typanalysen abgelehnt und konnten nur durch umfangreiche und sehr künstliche Veränderungen (alle Variablen mußten zu Beginn mit geeigneten Werten initailisiert werden) zu akzeptierbaren Programmen umgeformt werden. Daher haben wir bei den in dieser Arbeit vorgestellten Typanalysen ein eventuelles Verschwinden von uninitialiserten Variablen bewußt in Kauf genommen.

Kapitel 6

Das Tool SOLAT

Eine Typanalyse produziert eine große Menge an Typinformationen, die für Menschen schwer lesbar und verständlich sind. Eine Datenflußinformation an einem Knoten (Zustand) besteht aus der Abbildung von der Menge der Variablen in die Menge der Typen. Während der Arbeit mit SOLAT (Simple Object-oriented Language Analyzing Tool), das zunächst nur aus dem Analyse-Werkzeug bestand und eine einfache ASCII-Ausgabe produzierte, entstand die Notwendigkeit einer übersichtlicheren Darstellung der gewonnenen Typinformationen. Die ASCII-Ausgabe enthält zwar das annotierte Analyse-Modell des analysierten Programms, aber das Suchen nach einzelnen Programmpunkten und bestimmten Informationen erweist sich als sehr umständlich und zeitraubend. Daher ist SOLAT zu einem graphischem Werkzeug weiterentwickelt worden. Technische Details, wie Installierung und Konfigurierung von SOLAT sowie Beschreibung der Implementierung, können in [Po97] nachgelesen werden.

Da der graphische Teil der SOLAT-Implementierung in *[incr Tcl]/Tk* [Mc96] erfolgt ist, steht SOLAT für verschiedene Hardware-Plattformen zur Verfügung. Ausführlich getest worden ist SOLAT auf DEC Alpha-Stations, SUN Workstations und PC's unter Linux.

SOLAT erzeugt vier Arten von graphischen Fenstern (Haupt-, Identifier-, Graph-, Inspect-Fenster), die und ihre Benutzung im folgenden beschrieben werden. Nachdem Aufruf von SOLAT wird ein gegebenes Programm analysiert und ein Hauptfenster erzeugt, siehe 6.1.

6.1 Das Hauptfenster

Nach einigen statistischen Angaben (benötigte Analysezeit, Anzahl der analysierten Elemente der *workset*, Anzahl der Methodenaufrufe, Anzahl der mehrdeutigen Methodenaufrufe) wird im Hauptfenster das annotierte Analyse-Modell in ASCII-Ausgabe aufgelistet. Hierbei werden Variablen, vorangestellt mit dem Namen der sie

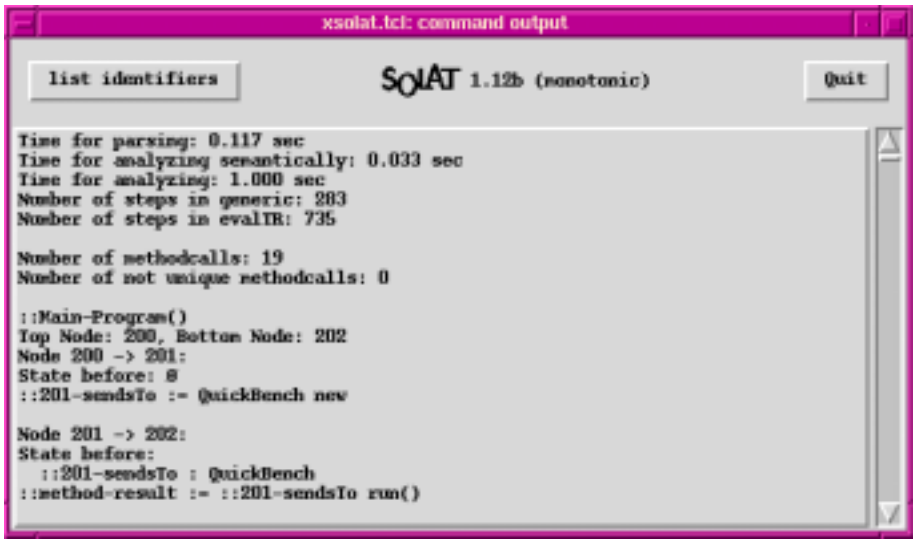


Abbildung 6.1: Das SOLAT Hauptfenster

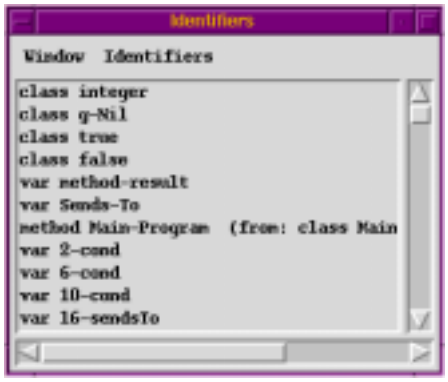
umfassenden Klasse und durch :: getrennt sowie nachfolgend und durch : getrennt mit ihrem momentan gültigen Typ versehen. D.h. $Test :: a : A$ bedeutet, daß die Variable a in der Klasse $Test$ vom Typ A ist. Bei Variablen, deren Klasse nicht explizit mit angegeben ist, handelt es sich um von der Analyse erzeugte Variablen, wie z.B. *e-sendsTo* oder *method-result*.

Ein Mausklick auf das list identifiers öffnet das Identifier-Fenster.

6.2 Das Identifier-Fenster

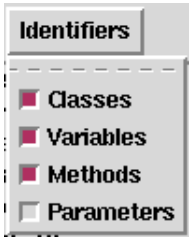
In diesem Fenster können alle im Programm vorkommenden Identifikatoren angezeigt werden, wobei auch eine Auswahl nur nach Klassen, Variablen, Methoden oder Parameter erfolgen kann.

Die Einträge im Identifier-Fenster können selektiert werden. Zuvor sollte mit Open new inspect window in das Windows Menü ein Inspect-Fenster erzeugt werden. In diesem



(a) Das Identifier-Fenster

(b) Das Window-Menü



(c) Das Identifier-Menü

Abbildung 6.2: Die Komponenten des Identifier-Fenster

Inspect-Fenster werden die Typinformationen für einzelne ausgewählte Identifikatoren dargestellt. Wie dies geschieht, erklären wir weiter unten. Wird nun ein Methodenidentifikator mittels der linken Maustaste selektiert, so wird im Hauptfenster das entsprechende annotierte Methodenmodell angezeigt. Beim Auswählen eines Methodenidentifikators durch die rechte Maustaste wird ein neues Fenster, das Graph-Fenster, geöffnet.

6.3 Das Graph-Fenster

Hier wird das Methodenmodell der ausgewählten Methode graphisch dargestellt, siehe Abbildung 6.3.

Der angezeigte Graph kann auch als Postscriptdatei abgespeichert oder gedruckt werden.

6.4 Das Inspect-Fenster

Nachdem aus dem Identifier-Fenster heraus ein Inspect-Fenster erzeugt worden ist, können im Identifier-Fenster Variablen- und Parameter-Identifikatoren ausgewählt

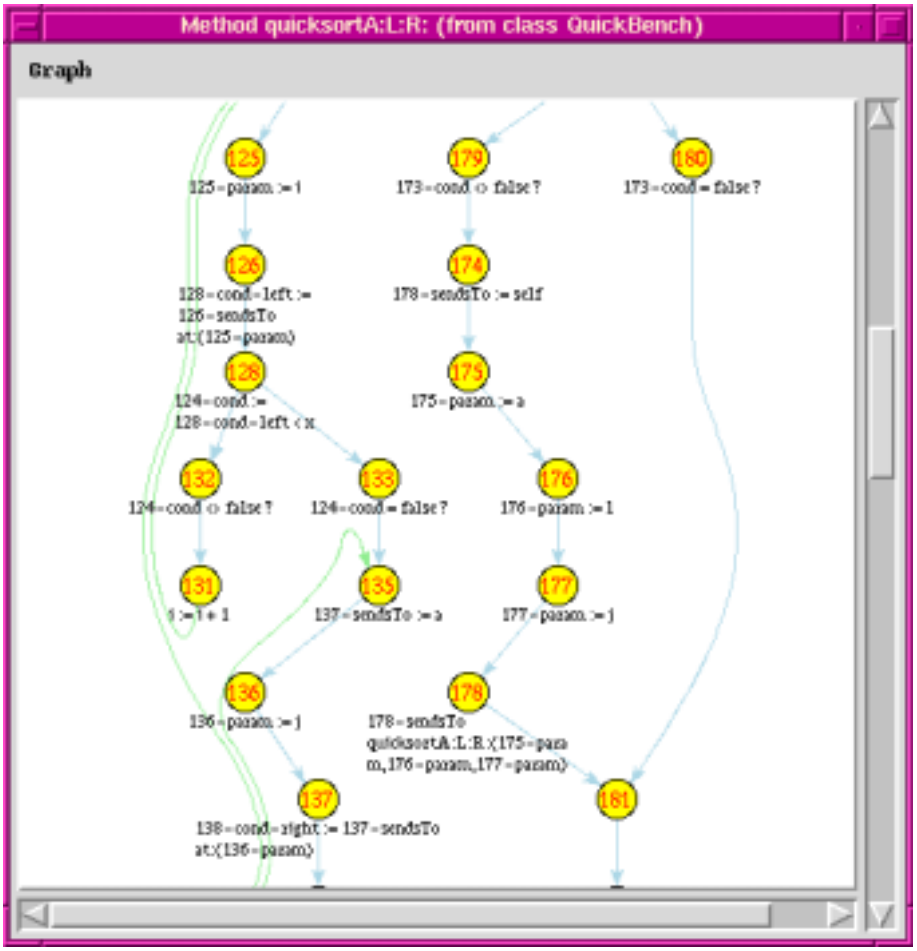
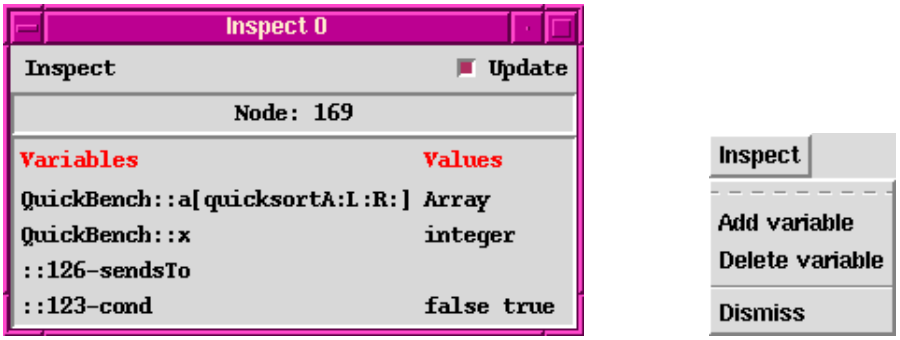


Abbildung 6.3: Das Graph-Fenster

werden, indem der Menüpunkt Open new inspect window im Inspect-Fenster aktiviert wird. Somit kann eine individuelle Auswahl von Variablen oder Parameter angelegt werden, die beobachtet werden können.

Wenn im Graph-Fenster mit der rechten Maustaste auf einen Knoten (Zustand) geklickt wird, so erscheinen im Inspect-Fenster die Typinformationen, die an diesem Knoten gültig sind, siehe Abbildung 6.4. Durch Invertieren des Update-Knopfes Update im Inspect-Fenster können die momentan angezeigten Informationen eingefroren werden. Es dürfen beliebig viele Inspect-Fenster angelegt werden.



(a) Das Inspect-Fenster (b) Das Inspect-Menü

Abbildung 6.4: Die Komponenten des Inspect-Fensters

6.5 Erfahrungen mit SOLAT

Die Benutzung von SOLAT bei der Programmentwicklung erwies sich als sehr hilfreich. Die Möglichkeiten nur einen Teil der Variablen zu beobachten sowie die graphische Darstellung der Methodenmodelle sind dabei ebenso hervorzuheben wie auch die aggressiven Typanalysen.

Es zeigte sich aber auch schnell eine Schwachstelle in der Definition der Typanalysen. Aus theoretischer Sicht ist die Verwendung der *SthOutOf*-Filter zur Modellierung der Fehlersituationen bei Methodenaufrufen naheliegend und begründet durch die dynamische Semantik. Aber wenn die benutzte Typanalyse eine solche Fehlersituation glaubt gefunden zu haben, so liefert der *SthOutOf*-Filter $\perp$ zurück. Dadurch werden alle weiteren Berechnungen geblockt und man kann über die weiteren Programmteile keine Aussagen machen. In der Praxis der Programmentwicklung haben wir dieses Verhalten als zu restriktiv kennengelernt. Wenn der *SthOutOf*-Filter einen möglichen Fehler meldet, so ist es meistens sehr aufwendig die genaue Fehlerstelle zu lokalisieren. Um dies zu vereinfachen, ist SOLAT mit einer Option versehen worden, daß die *SthOutOf*-Filter nicht berücksichtigt werden. Dies ist in der momentan aktuellen SOLAT-Version sogar der Normalfall (default). In den formalen Definitionen werden die *SthOutOf*-Filter ersatzlos gestrichen. Stattdessen wird nach Ende der Analyse der gesamte Programmgraph nochmals durchlaufen, und jeder Methodenaufruf wird daraufhin überprüft, ob der Empfänger und sein berechne-

ter Typ den entsprechenden Methodenaufruf zulassen. Wenn nicht, dann wird eine Fehlermeldung am Anfang der Ausgabe der Typanalyse produziert, bei der die "fehlerbeladene" Knotennummer mit ausgegeben wird. Gibt es in einem Programm mehrere Fehlersituationen, so werden diese alle mit aufgeführt und nicht nur der erste aufgetretene Fehler. In dieser Form kann die Typanalyse in einen Compiler eingebaut werden. Der Compiler kann Warnungen ausgeben und trotzalledem die korrekten Methodenaufrufe optimieren. Man beachte stets, daß nicht alle von der Typanalyse als fehlerhaft erkannten Methodenaufrufe zur Laufzeit auch wirklich Fehler produzieren. Somit ist die Verwendung der *SthOutOf*-Filter aus theoretischer Sicht durchaus wünschenswert, aber aus praktischer Sicht ungeeignet.

Eine andere Erfahrung bei der Arbeit mit SOLAT ist gewesen, daß die aggressiveren (präziseren) Typanalysen defensives Programmieren erzwingen, da sonst Fehlermeldungen produziert werden. Vor allem die Initialisierung von Variablen haben bei fast allen Testprogrammen, die zunächst von Smalltalk-80 nach SOL übersetzt worden sind, zu Schwierigkeiten geführt. Häufig haben Variablen in Verzweigungen oder Schleifen erstmalig einen Wert zugewiesen bekommen. Eine Initialisierung vor den Verzweigungen bzw. Schleifen ohne semantische Änderungen des Programmes sorgt für akzeptierbare Programme. Innerhalb der Software-Entwicklung ist dies ein Charakteristikum für defensives Programmieren.

Kapitel 7

Verwandte Ansätze

Statische Programmanalysen sind ein seit 20 Jahren stetig wachsender Forschungsbereich. Die meisten neueren Ansätze stützen sich immer noch auf die "alten" grundlegenden Arbeiten zur Datenflußanalyse [KU77, He77, MJ81, SP81], Abstrakten Interpretation [CC77, CC79] oder Typinferenz [Hi69, Mi78]. Allein der Entwicklung der (intra- bzw. interprozeduralen) Datenflußanalyse könnte ein längeres Kapitel gewidmet werden. Wir wollen uns aber im vorliegenden Kapitel vor allem auf die Ansätze konzentrieren, die zur Berechnung von Typinformationen in objekt–orientierten Sprachen verwendet werden. Hierbei betrachten wir drei grundsätzlich verschiedene Vorgehensweisen (Typinferenz, Datenflußanalyse, Type-Feedback). Abgerundet wird der Überblick mit einigen Arbeiten zur Typanalyse in anderen Sprachen.

7.1 Typinferenz

Zunächst werden drei "ältere" Typinferenzansätze vorgestellt, die alle auf der Hindley/Milner Typinferenz für funktionale Sprachen basieren. Alle drei Ansätze berücksichtigen keinerlei Datenfluß und die Analyse der Methoden erfolgt intraprozedural. Danach präsentieren wir vier Ansätze der 90er Jahre, bei denen dynamische Bindung der Methodenaufrufe durch *bedingte Constraints* modelliert wird und anstelle von Unifikation nun Fixpunktableitungen zur Berechnung der Lösungen des Constraints-Systems ausgeführt werden. Eine ausführlichere Beschreibung der Ansätze zur Typbestimmung in objekt–orientierten Sprachen ist in [Ag95b] zu finden.

Hindley und Milner

Die Mutter (oder Vater) aller Typinferenz-Algorithmen stellt die Arbeit von Hindley [Hi69] dar. Er entwickelte eine Typinferenz für die kombinatorische Logik und für den Lambda–Kalkül. Dieser wurde später von Milner [Mi78] auf Programmiersprachen erweitert. Dieser Algorithmus ist heute noch (in modifizierter Form) Bestandteil eines jeden ML–Compiler.

In dem Typinferenz–Modell existieren drei Arten von Typen: *Basistypen* für primitive Objekte, *Typvariablen* zur Behandlung von polymorphen Typen und *Funktionstypen* (Argumenttyp $\rightarrow$ Resultattyp). Typinferenz wird ausgeführt, indem der Ausdrucksbaum (expression tree) von den Blättern zur Wurzel hin (bottom-up) analysiert wird. Dabei werden zunächst die Blätter mit ihren entsprechenden Typen versehen (Blatt ist Konstante: Basistyp; Blatt ist Variable: Typvariable) und danach die Typen mittels Unifikation berechnet. Die charakteristischen Merkmale der Hindley/Milner-Typinferenz im Hinblick auf unsere Typanalyse sind:

- Die Hindley/Milner-Typinferenz berechnet den am meisten allgemeinen oder prinizipiellen Typen (most general type) und nicht den möglichst exaktesten Typ. Die Typinferenz nimmt zu Anfang an, daß Typen so allgemein wie möglich sind, und durch die Analyse erst eingeschränkt werden. Unsere interprozeduralen Analysen gehen von dem umgekehrten Standpunkt aus, zu Anfang wird jede Variable mit dem kleinsten Typ versehen.

- Die Hindley/Milner-Typinferenz berechnet jede Funktion allein, d.h. intraprozedural.

- Die Hindley/Milner-Typinferenz ist flußinsensitiv (flow-insentive), d.h. verschiedene Zugriffe auf eine Variable sind vom gleichen Typ.

Die Hindley/Milner-Typinferenz ist später um weitere Sprachkonstrukte ausgebaut worden [To90, Ca88]. Hense und Smolka definieren eine Typinferenz für objekt–orientierte Sprachen mit Vererbung [He93].

Borning und Ingalls

Einen der ersten Ansätze zur Berechnung von Typinformationen für Smalltalk-80 haben Borning und Ingalls in [BI82] vorgestellt. Sie entwickelten und implementierten ein Typinferenzsystem für Smalltalk-80. Dabei müssen alle Variablen, formalen Parameter und Resultate von Methoden mit einem statischen Typ deklariert werden. Jede Methode wird einzeln analysiert. Weiterhin können durch die Analyse von Verzweigungen zwei unterschiedliche Typen nicht vereint werden, sondern anstelle dessen wird die nächste gemeinsame Superklasse beider Typen als Typ verwendet. Z.B. seien A, B, C Klassen in einem Programm und A, B sind Teilklasse von C. Die Analyse des Ausdrucks `x=y ifTrue:[A new] ifFalse:[B new]` liefert nicht den Typ $\{A, B\}$ sondern $\{C\}$ zurück. Borning und Ingalls testeten ihr System an einem kleinen Teil des Smalltalk-80–Systems. Allerdings sind keinerlei Ergebnisse über den Erfolg oder Mißerfolg ihrer Tests publiziert worden.

Suzuki

Ungefähr zeitgleich mit Borning und Ingalls entwickelte Suzuki [Su81] ein Typinferenzsystem für Smalltalk-76. Dabei ist erstmals ein Typ eine Menge von Klassen. Suzuki erweiterte Hindley/Milner-Typinferenz um objekt–orientierte Konstrukte (Zuweisungen und dynamisch gebundene Methodenaufrufe). Hierbei trat folgendes zirkuläre Problem auf:

- Um den Typ eines Methodenaufrufes abzuleiten, muß die Menge der potentiellen Methodendefinitionen bekannt sein.
- Um die Menge der potentiellen Methodendefinitionen zu bestimmen, muß der Typ des Empfängers des Methodenaufrufes, möglicherweise wiederum ein Methodenaufruf, bekannt sein.

Diese gegenseitige Abhängigkeit wird derart aufgelöst, daß zu Beginn der Analyse alle Methodendefinitionen gleichen Namens als mögliche Kandidaten angenommen werden. Um diese pessimistische (maximal conservative) Annahme zu verbessern, wurde die Typinferenz wiederholt angewandt, sofern im vorherigen Iterationsschritt eine weniger pessimistische Typinformation berechnet wurde. Ähnlich wurden auch

die Typen von Variablen bestimmt. Dieses Verfahren kann zu einer Überspezifikation der Typen führen, da alle Methoden zu Beginn analysiert werden, auch die, die niemals zur Laufzeit ausgeführt werden.

Suzuki implementierte eine vereinfachte Version seiner Typinferenz, aber Ergebnisse über eine Anwendung der Typinferenz sind nicht bekannt.

Graver und Johnson

Im Typed Smalltalk-Projekt [GJ90, Gr89, Jo86, JGZ88] sollte Smalltalk-Code durch Hinzunahme eines statischen Typsystems verbessert werden. Instanz-, Klassen- und globale Variablen mußten mit einem Typen deklariert werden. Lokale Variablen, formale Parameter und Methoden konnten, mußten aber nicht, mit Typdeklarationen versehen werden. Aufgabe der Typinferenz war, für die ungetypten Programmteile geeignete Typen zu berechnen. Dabei wurden drei Arten von Typen unterschieden: *Klassentypen* (z.B. $Integer$), *Vereinigungstypen* (z.B. $\{Integer, Float\}$) und *Signaturtypen* (z.B. $Self\ METH\ :\ Integer \rightarrow Float$, wobei $Self$ den Typ des Empfängers der Methode $METH$, $Integer$ den Argumenttyp und $Float$ den Resultattyp bezeichnen). Vererbung wird durch Kopieren eliminiert. Nach Konstruktion des TypeTrees einer Methode aus dem ParseTree durch geeignete Abstraktion wird die Methode abstrakt interpretiert.

Die Kombination von Typdeklaration und Typinferenz erzeugt folgendes Dilemma:

- Typdeklarationen sollen so abstrakt wie möglich sein, um Wiederverwendbarkeit von Programmcode zu unterstützen.

- Typinferenz soll so konkret (präzise) wie möglich sein, um Optimierungen zu unterstützen.

Dies bedeutet für den Programmierer, daß je allgemeiner seine Deklarationen sind, desto weniger Optimierungsmöglichkeiten hat der Compiler.

Der Nutzen der Implementierung des Typinferenzsystems wird in [JGZ88] nur anhand von recht kleinen Programmen illustriert, sodaß eine echte Einschätzung nicht erfolgen kann.

Oxhøj, Palsberg und Schwartzbach

Das Typinferenzsystem von Palsberg und Schwartzbach [PS91] behandelt Programme einer einfachen objekt–orientierten Sprache, im wesentlichen identisch zu SOL . Ihr Ansatz basiert auf folgende Überlegungen:

1. Vererbung wird durch Kopieren eliminiert.
2. Typen sind Mengen von Klassen.
3. Jeder Variablen v und jedem Ausdruck e wird eine Typvariable zugeordnet $(v \rightarrow [\![v]\!], e \rightarrow [\![e]\!])$.
4. Bilde das Programm auf die Menge von bedingten Constraints der Form ab $[\![a]\!] \subseteq [\![b]\!]$ bzw. $[\![a]\!] = [\![b]\!]$ bzw. $C \in [\![a]\!] \Rightarrow [\![b]\!] \subseteq [\![c]\!]$, wobei a, b, c Variablen, formale Argumente oder Ausdrücke sind und C eine Klasse ist.
5. Löse dieses Constraints-System durch eine Fixpunktberechnung, die in polynomieller Zeit durchgeführt werden kann, siehe [PS93].
6. Annotiere das analysierte Programm mit den Informationen der Lösung des Constraints–Systems.

Da dieser Ansatz Grundlage für die weiteren Typinferenzsysteme ist, werden wir die wesentlichen Vorgehensweisen in 4. genauer erläutern. Dazu verwenden wir, wie in [Ag94, Ag95a, Ag95b], folgende Anschauung:
Es wird ein gerichteter Graph aufgebaut, dessen Knoten Typvariablen beinhalten. Die Kanten, die nach und nach hinzugefügt werden, repräsentieren die Constraints, durch die der Laufzeitdatenfluß modelliert wird. Z.B. bewirkt die Zuweisung $x := e$ das Hinzufügen einer Kante von $[\![e]\!]$ nach $[\![x]\!]$. Die Kante repäsentiert also das Constraint $[\![e]\!] \subseteq [\![x]\!]$, siehe Abbildung 7.1.

Zu Beginn der Analyse liegt ein unstrukturierter Graph vor bestehend aus der Menge der Typvariabalen. Dieser Graph läßt sich in Teilgraphen einteilen, wobei jeder Teilgraph genau einer Methode entspricht. Diese Teilgraphen werden auch Templates genannt. Ein Template einer Methode M besteht aus

- den Typvariablen der Ausdrücke, lokalen Variablen und formalen Parameter von M und

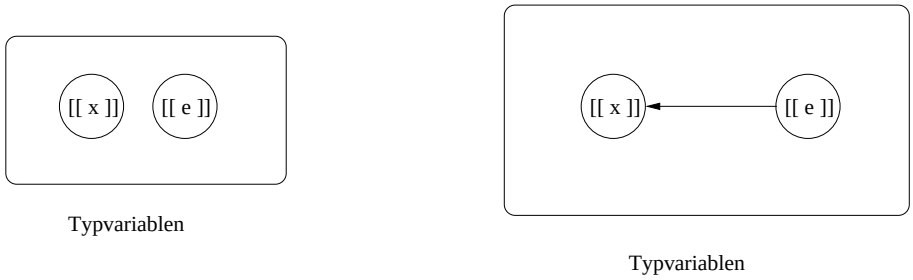


Abbildung 7.1: Nach Ausführung von $x := e$ wird die Typinformation von $[\![e]\!]$ nach $[\![x]\!]$ propagiert

- den Constraints der Typvariablen.

Knoten, die Instanzvariablen repräsentieren, sind nicht Teil eines Templates, sondern werden außerhalb der Templates verwaltet.
Betrachten wir die Methode

```
method min:arg
   (self<arg) ifTrue:self ifFalse:arg
```

und den Methodenaufruf `(A new) min:(A new)`. Abbildung 7.2 zeigt das Template der Methode `min:`. Die Auswertung von `self<arg` liefert den Typ $\{false, true\}$ zurück, wodurch das `min:`-Template mit den `ifTrue:ifFalse:`-Templates der Klassen $false$ und $true$ verbunden ist. Ebenso wird der Aufruf `(A new) min:(A new)` mit dem `min:`-Template verbunden. Der Typinferenz–Algorithmus propagiert vom Methodenaufruf ausgehend Typinformationen durch den Graphen und errechnet, daß der Resultattyp der Methode `min:` $\{A\}$ ist.

Der Typinferenz–Ansatz in [PS91] ordnet jeder Methodendefiniton genau ein Template zu. Somit werden verschiedene Aufrufe von `min:` mit dem gleichen Template verbunden. Dies kann zu Ungenauigkeiten der Typinformationen führen, wie in Abbildung 7.3 zu sehen ist. Damit wäre nach Analyse der Zuweisungen

$$\texttt{a} := (\texttt{A new})\ \texttt{min} : (\texttt{A new}) \quad \textit{und} \quad \texttt{b} := (\texttt{B new})\ \texttt{min} : (\texttt{B new})$$

sowohl die Variable `a` als auch die Variable `b` vom Typ $\{A, B\}$.

Dieses Problem haben Oxhøj, Palsberg und Schwartzbach in ihrer Implementierung [OPS92] berücksichtigt. Die Typinferenz wurde derart verändert, daß für jedes syntaktische Vorkommen eines Methodenaufrufes ein eigenes Template erzeugt wird.

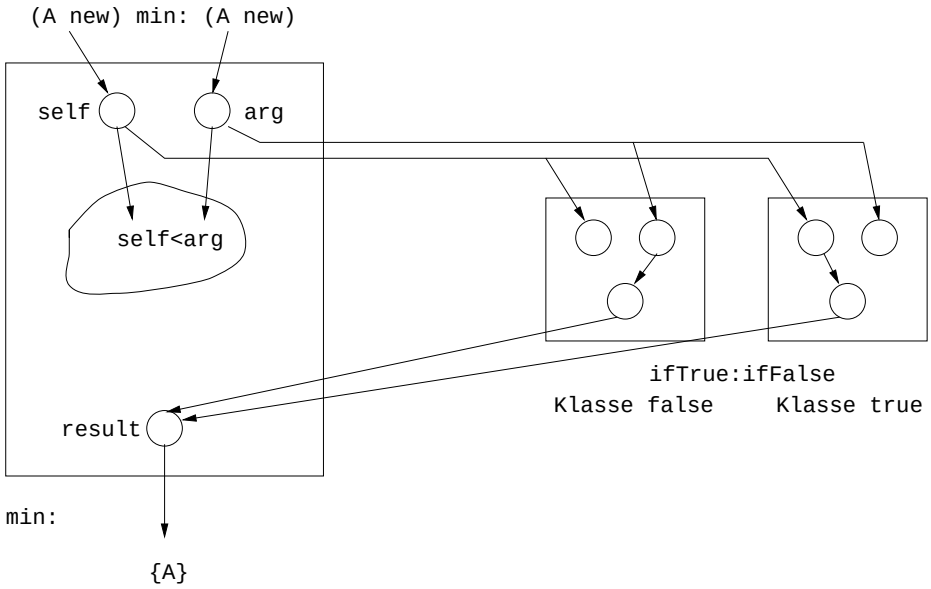


Abbildung 7.2: Template für die Methode `min:` und einem Aufruf, der mit `min:` verbunden ist.

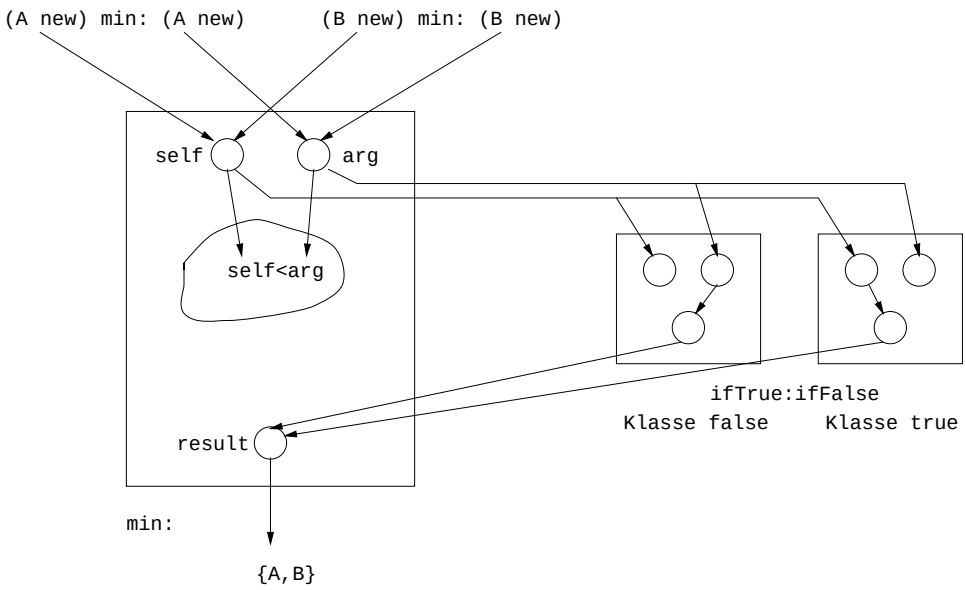


Abbildung 7.3: Template für die Methode `min:` und zwei Aufrufe, die mit `min:` verbunden ist.

Hierdurch wird allerdings das gesamte Programm erheblich vergrößert (quadratisch in der Größenordnung des Programms). Dieser polyvariante Ansatz, auch 1-level Typinferenz genannt[1], kann in manchen Fällen zur Verbesserung der Typinformationen führen, aber im obigen Beispiel in Abbildung 7.4 verändert sich nichts.

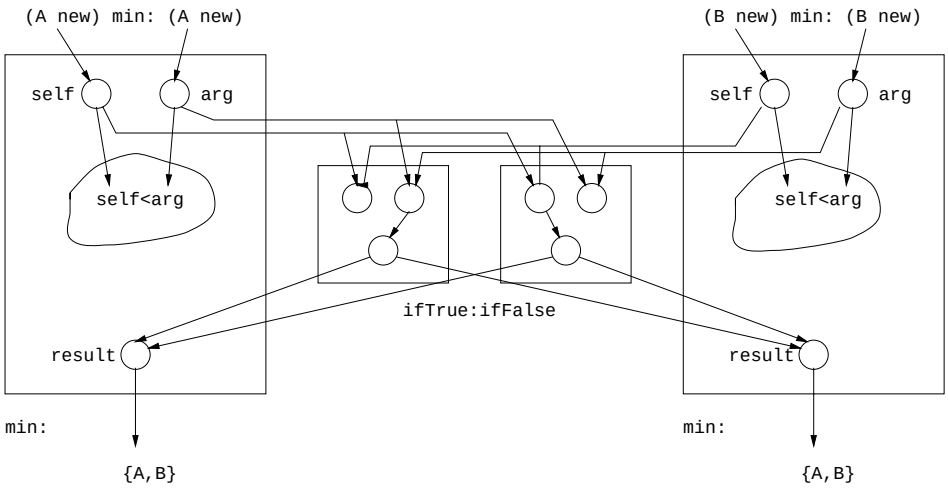


Abbildung 7.4: 1-level Typinferenz.

Die 1-level Typinferenz wurde an kleinen Programmen getestet, die wir in unserer Testsuite in Kapitel 5 aufgenommen haben. Die Behandlung von Zuweisungen und Verzweigungen entspricht unserer Typanalyse PS. Allerdings verwendet die Typanalyse PS für jede Aufrufstelle ein separates Template, genauer Activation Record.

Phillips und Shepard

Der 1-level Typinferenzansatz von Oxhøj, Palsberg und Schwartzbach wurde von Phillips und Shepard in [PS94] auf beliebigen p-level verallgemeinert. Bei ihrer Implementierung kann der Benutzer den Level der Analyse vorgeben. Ihr p-level Typinferenz–System wurde in Smalltalk-80 implementiert und getestet. Die Laufzeitergebnisse der Tests waren verhehrend gewesen. Die Analyse von einfachen Ausdrücken benötigte 30 min für die 0-level Typinferenz und bis zu 10 Stunden für die 1-level Typinferenz. Ihre Implementierung ist zwar nicht hochoptimierend gewesen, aber dennoch muß die reale Einsetzbarkeit ihres Systems bezweifelt werden.

[1]Entsprechend wird der vorherige monovariante Ansatz 0-level Typinferenz genannt.

Plevyak und Chien

Die Ungenauigkeiten des 1-level Typinferenzansatzes umgingen Plevyak und Chien in ihrem Ansatz [PC94, PC97, PC95a, PC95b] durch iterierte Anwendung der Typinferenz. Zunächst wird ein Programm mit der 1-level Typinferenz analysiert. Dann werden die Stellen im Programm, bei denen Ungenauigkeiten in den Typinformationen auftreten, durch geeignete Erweiterungen modifiziert. Danach wird das (modifizierte) Programm erneut mit der 1-level Typinferenz analysiert. Dieser Vorgang wird solange wiederholt, bis die Typinformationen stabil sind. Bei den Erweiterungen handelt es sich im wesentlichen um geschicktes Aufsplitten, Vervielfachen (Verfeinern) von Templates, siehe Abbildung 7.2.

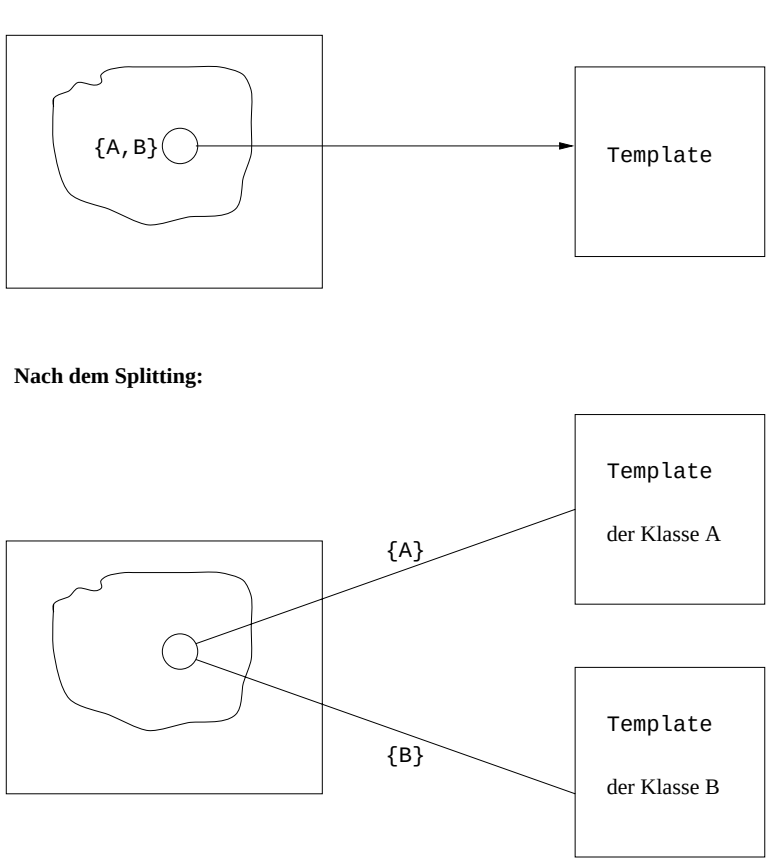


Abbildung 7.5: Splitten von Templates bei unscharfen Typinformationen.

Angenommen ein Knoten beinhaltet einen unscharfen Typen und dieser Knoten ist mit einem anderen Template verbunden. Dieses Template wird nun vervielfacht und der Knoten auch mit den neuen Templates verbunden, siehe Abbildung

7.5. Bei der nachfolgenden Analyse erhalten alle diese Templates eindeutige Typinformationen. Die Bestimmung der unscharfen Programmstellen ist nichttrivial und bedarf aufwendiger Untersuchungen des Graphen. Plevyak und Chien konnten damit nicht nur Funktionenpolymorphismus auflösen, sondern auch Datenpolymorphismus (Problem der Containerklassen) in den Griff bekommen. Im Vergleich zu unseren Typanalysen, die Datenpolymorphismus nur in sehr eingeschränkter Form behandeln können, ähnelt ihr Ansatz unserer distributiven Typanalyse, wenn wir die Qualität der Typinformationen zugrundelegen. Auf die deterministische Behandlung von Verzweigungen wird in ihrer Typinferenz verzichtet. Plevyak und Chien haben ihren Typinferenz–Algorithmus für die Sprache Concurrent Aggregate implementiert. Leider ist diese Sprache mit SOL nicht vergleichbar, sodaß die erzielten Resultate mit unseren Ergebnissen nicht in Beziehung gesetzt werden können.

Ageson

In [Ag95a, Ag95b] stellt Ageson seine kartesiches-Produkt-Typinferenz vor (cartesian product algorithm, CPA). Dieser arbeitet ähnlich der 1-level Typinferenz mit Ausnahme der Behandlung von Methodenaufrufen. In der CPA-Typinferenz wird für jede Kombination des Empfängertyps und der Argumenttypen ein eigenes Template angelegt. D.h. wenn `revr meth:arg` ein Methodenaufruf ist und `revr` vom Typ $\{R_1, ..., R_k\} =: R_T$ sowie `arg` vom Typ $\{A_1, ..., A_k\} =: A_T$ sind, dann werden $|R_T \times A_T|$ verschiedene Templates zur Analyse des Methodenaufrufes angelegt, siehe Abbildung 7.6.

Dabei wird jedes Paar $(r_i, a_j) \in R_T \times A_T$ einem separaten Template übergeben. Wenn ein solches Template bereits existiert, so wird das alte wiederverwendet, ansonsten ein neues erzeugt. Die zugrundeliegende Idee ist sehr einfach: eine Variable (oder Parameter) hat zur Laufzeit nur einen eindeutigen Typen. Unabhängig von Ageson wurde in [Go95] diese Möglichkeit ebenfalls angesprochen.

Im Gegensatz zum iterativen Verfahren von Plevyak und Chien kommt die CPA-Typinferenz ohne Iterationen aus. Trotz polynomieller Komplexität kann bei großer Anzahl von Argumenten die Laufzeit extrem anwachsen. Ein Methodenaufruf mit 18 Argumenten, die jeweils vom Typ $\{A, B, C\}$ sind, erzeugt $3^{19} = 10^9$ Templates, die analysiert werden müssen. Aber bei allen von Ageson getesteten Programmen

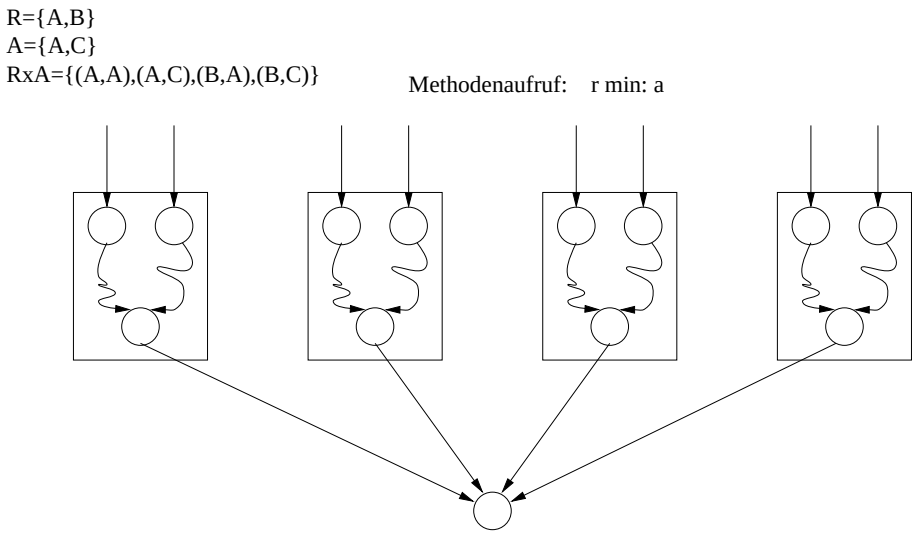


Abbildung 7.6: Kartesisches Produkt Typinferenz.

trat ein solches Verhalten nicht auf. Die Laufzeiten der Programme, z.T. ähnlich unserer Testsuite, sind allesamt gut gewesen. Da die Programme in der Sprache Self [US91] geschrieben sind und Self eine Prototyp–basierte objekt-orientierte Sprache mit dynamischer Vererbung ist, lassen sich die erzielten Ergebnisse mit unseren Resultaten nicht direkt vergleichen. Das kartesiche Produkt wird bei der distributiven Typanalyse DIST implzit auch verwendet. Die CPA–Typinferenz ist weniger präzise als unsere distributive Typanalyse und z.T. präziser als unsere monotonen Typanalysen. Die deterministische Behandlung von Verzweigungen fehlt ebenso wie destruktive Zuweisungen. Letztere behandelt Ageson in [Ag95b], hat sie aber nur für lokale Variablen implementiert.

7.2 Datenflußanalyse

Neben unseren interprozeduralen Datenflußanalysen gibt es noch weitere Datenflußansätze [Ba92, VHU92, Gr95]. Die klassische interprozedurale Datenflußanalyse analysiert den Aufrufgraphen (call graph) eines Programmes. Die Vorgehensweise kann auf objekt–orientierte Sprachen nicht unmittelbar übertragen werden.

Barnard

In [Ba92] vergleicht Barnard eine vereinfachte Version der Graver/Johnson–Typinferenz mit einer von ihm definierten Datenflußanalyse. Er erkannte, daß Kontrollfluß und Datenfluß voneinander abhängig sind. Seine theoretischen Untersuchungen ergaben, daß interprozedurale Datenflußanalyse diejenigen fehlerkritischen Programmpunkte ermitteln kann, die von der Typinferenz erkannt werden. Darüberhinaus vermeidet die Datenflußanalyse im Gegensatz zur Typinferenz die Ablehnung manchen "korrekten" Codes. Die theoretischen Modelle wurden nicht implementiert.

Vitek, Horspool und Uhl

Eine weitere interprozedurale Datenflußanalyse für eine einfache objekt–orientierte Sprache (vergleichbar mit SOL) präsentieren Vitek, Horspool und Uhl in [VHU92]. Ihre Analyse ist eine verbandsbasierte Datenflußanalyse, deren Verband abstrakte Objektgraphen modelliert. Ein abstrakter Objektgraph approximiert den Zustand des Laufzeit–Heaps (Objektspeicher, Objectmemory in Smalltalk-80) an einem Programmpunkt. Darin werden abstrakte Objekte und Verbindungen von Instanzvariablen zu abstrakten Objekten dargestellt. Mittels Fixpunktiteration werden die Typinformationen berechnet, sodaß jeder Programmpunkt mit einem abstrakten Objektgraphen annotiert werden kann. Um die Analyse zweier Aufrufstellen einer Methode überdecken zu können, verwenden Vitek, Horspool und Uhl das *call-string*-Konzept. Ein call string für einen Programmpunkt besteht aus der Folge der aufgerufenen Methoden auf dem Pfad vom Hauptprogramm zu dem Programmpunkt hin. Da Pfade beliebig lang sein können, werden nur die letzten p Methodenaufrufe im call string aufgenommen. Sind zwei call strings identisch, so wird die Analyse der Methode an der zweiten Programmstelle auf die Ergebnisse der Analyse der ersten Programmstelle zurückgeführt. Call strings der Länge 1 entsprechen dem 1-level der 1-level Typinferenz. Da die Vitek, Horspool und Uhl Datenflußanalyse nicht implementiert worden ist, kann über die Güte des Verfahrens nur spekuliert werden.

Chambers, Dean und Grove

Grove präsentiert in [Gr95] Ergebnisse über eine interprozedurale Datenflußanalyse, die im Rahmen des Vortex–Projektes [DDGLC96, CDG96a, CDG96b] (effiziente Implementierung der objekt–orientierten Sprache Cecil) entstanden ist. Seine Datenflußanalyse konstruiert und analysiert den Aufrufgraphen eines Programmes simultan. Die Zwischenergebnisse der Analyse werden für den weiteren Aufbau des Aufrufgraphen verwendet. Wenn die Informationen, die bei zwei Aufrufstellen der gleichen Methode vorliegen, identisch sind, wird die Analyse der Methode nur einmal ausgeführt. Ansonsten entspricht die Grove–Datenflußanalyse der 1-level Typinferenz. Damit gleicht sie unserer Typanalyse PS, wobei die Struktur der Typinformationen und ihrer Verwaltung anders sind.

Das interessanteste Ergebnisse der Untersuchungen von Grove ist der erzielte Nutzen einer solchen Analyse. Es wurden Programme mit bis zu 45000 Zeilen (lines of code) analysiert. Aufgrund der Typinformationen konnten Optimierungen vorgenommen werden, die Laufzeitverbesserungen von teilweise weit über 25% (bis zu 144%) gegenüber Standardtechniken (intraprozedurale Analysen) erreichten. Dies unterstreicht die Bedeutung von interprozeduraler Datenflußanalyse. Weitere Verbesserungen können durch den Einsatz präziserer Typanalysen erwartet werden.

7.3 Type-Feedback

Ein rigeros anderer Ansatz zur Gewinnung von Typinformationen ist im momentanen Self–Compiler integriert.

Hölzle

Der Schlüssel zur Berechnung von Typinformationen ist die Verwendung von Informationen, die erst zur Programmlaufzeit gewonnen werden. Eine Methode wird zunächst ohne Optimierung übersetzt, wobei im Methodencode Platz für *profile*–Informationen freigehalten wird. Die Häufigkeit der Ausführung einer Methode und die jeweiligen Empfänger werden mitprotokolliert. Man bedenke, daß Self dynamische Vererbung unterstützt und somit eine Eliminierung der Vererbung nicht möglich ist. Wenn nun eine Methode zum x-ten Mal aufgerufen wird, wird der unoptimierte

Code auf der Grundlage der gesammelten Typinformationen optimiert, siehe Abbildung 7.7.

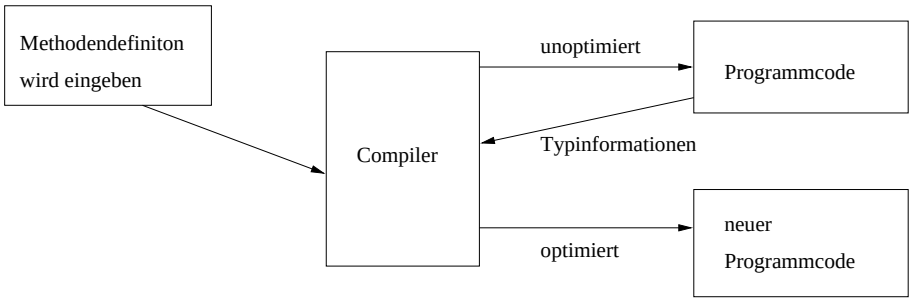


Abbildung 7.7: Type-Feedback.

Die erzielten Resultate [HU94, Hö95] zeigen eine signifikante Laufzeitverbesserung der optimierten Programmteile (einen Faktor von durchschnittlich 1.7) gegenüber der unoptimierten Version.

Diese Art der Typgewinnung berechnet eine untere Schranke der tatsächlich auftretenden Typen zur Laufzeit, wohingegen alle statischen Analysen eine obere Schranke bestimmen, siehe Abbildung 7.8.

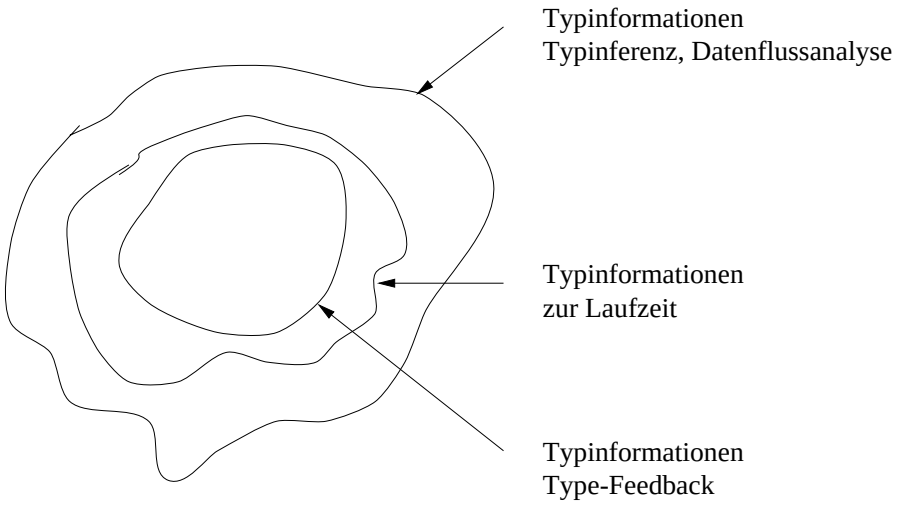


Abbildung 7.8: Grenzen der verschiedenen Verfahren.

In [AH95] haben Ageson und Hölzle ihre beiden unterschiedlichen Ansätze verglichen. Beide Verfahren liefern ähnlich gute Resultate, wobei der Typinferenz–Ansatz bei der Laufzeit leichte Vorteile hat (ca 15%). Dafür ist der Type–Feedback–Ansatz für interaktives Programmieren wesentlich besser geeignet. Während der Pausen

eines Programmierers kann quasi unbemerkt die optimierende Compilation durchgeführt werden.

7.4 Weitere Ansätze

An dieser Stelle seien noch ein paar Verfahren erwähnt, die für andere Programmiersprachen entwickelt wurden.

Shivers [Sh91b, Sh91a] entwickelte eine Berechnung der Kontrollflußgraphen für Scheme–Programme, um Datenflußanalysen anwenden zu können. Die Behandlung von Funktionen höherer Ordnung in funktionalen Sprachen und die Behandlung von dynamisch gebundenen Methodenaufrufen zeigen gewisse Parallelitäten. Beide Aufrufarten hängen von ihrer aktuellen Umgebung ab. Shivers startet mit einer denotationellen Semantik in cps-Form (continuation passing style) und leitet davon eine Nichtstandard–Semantik ab, die den relevanten Kontrollfluß modelliert. Da die Nichtstandard–Semantik nicht berechenbar ist, wird diese weiter modifiziert (abstrahiert). Shivers präsentiert verschiedene Abstraktionen mit unterschiedlicher Präzision und Analysezeit. Der einfache Algorithmus (OCFA) entspricht in etwa der 0-level Typinferenz und der 1CFA-Algorithmus entspricht in etwa der 1-level Typinferenz. Die Güte von Shivers–Ansatz kann aus praktischer Sicht nicht beurteilt werden, da nur sehr kleine Programme getestet worden sind.

Ebenfalls mit Typinferenz in funktionalen Sprachen beschäftigen sich Aiken, Wimmers und Lakshman [AWL94]. Sie definieren bedingte Typen zur Behandlung von case–Ausdrücken, um die geeigneten Zweige selektieren zu können. Die bedingten Typen entsprechen im wesentlichen unserer Filtertechnik zur genauen Analyse von Methoden und Verzweigungen. Allerdings sind in ihrer funktionalen Sprache keine Zuweisungen erlaubt, wodurch sich die Typinferenz bemerkbar vereinfacht. Pundt hat in [Pu97] den hier vorgestellten Datenflußanalyse-Rahmen auch für funktionale Sprachen angepaßt und versucht Bindungen von funktionalen Argumenten und Resultate sowie überflüssige Updates zu optimieren.

Typanalysen für C++-Programme ist ein aufstrebendes Gebiet der aktuellen Forschung [Pa96, PR94, CSH95]. Klassische Datenflußanalysen, wie z.B. code motion, können in C++–Programmen aufgrund von virtuellen Aufrufen (virtual calls,

entsprechen dynamisch gebundenen Methodenaufrufen) nicht so erfolgreich wie in C–Programmen ausgeführt werden, obwohl relativ wenige virtuelle Aufrufe in einem C++–Programm vorkommen. Daher sind Typanalysen sinnvoll, um einen verbesserten Kontrollflußgraphen konstruieren zu können. In C++–Programmen werden häufig Zeiger verwendet, sodaß eine Typanalyse von einer Pointer- und einer alias-Analyse nicht getrennt werden kann. In den Arbeiten von Pande und Ryder [Pa96, PR94] und von Carini, Srinivasan und Hind [CSH95] werden solche kombinierten Analysen vorgestellt.

Abschließend möchten wir noch auf eine andere Analyse hinweisen, der *static class hierarchy analysis*, die von Dean, Grove und Chambers [DGC95] vorgestellt wurde. Die Klassenhierarchie wird nach Situationen durchsucht, in denen Methoden in ihren Teilklassen nicht überschrieben (overloaded) werden. Wenn der Empfänger einer Methode m vom Typ S ist und S die Teilklasse einer Klasse C ist, die wiederum eine Definition von m enthält, wobei die Methode m in keiner Teilklasse von C überschrieben wird, so kann der Methodenaufruf statisch gebunden werden. Beim Vergleich mit Type–Feedback hat die Type–Feedback–Technik bzgl. der Laufzeit eindeutige Vorteile hat. Eine Kombination beider Techniken erzielte die besten Resultate, sodaß der static class hierarchy analysis Ansatz durchaus bemerkenswerte Verbesserungen bewirken kann.

Kapitel 8

Schlußbemerkungen

In dieser Arbeit ist ein Konzept zur Datenflußanalyse in objekt–orientierten Programmen vorgestellt worden. Das Konzept eignet sich zur Definition verschiedenster Datenflußanalyseprobleme, wie z.B. code motion, reaching definitions, constant propagation, dead code elimination, very busy expressions, Typanalyse. Anhand von letztgenanntem Analyseproblem haben wir die Anwendbarkeit und Realisierbarkeit des vorliegenden Konzeptes evaluiert und fünf unterschiedliche Typanalysen untersucht. Die Typanalysen sind einerseits theoretisch bzgl. Mächtigkeit und Komplexität und andererseits praktisch bzgl. verschiedener Kriterien wie Analysezeit, Qualität der gewonnenen Typinformationen und Speicherbedarf, untersucht und verglichen worden. Die aufgrund aller Ergebnisse favorisierte Typanalyse MONO ist ein guter Kompromiß zwischen Theorie und Praxis. Theoretisch ist sie zwar nicht die beste, aber in der Praxis gut verwendbar.

Über den Rahmen einer Datenflußanalyse hinaus ist das Werkzeug SOLAT vorgestellt worden, das die berechneten Typinformationen graphisch darstellen kann und zur Unterstützung der Programmentwicklung benutzt werden kann.

Ein Grund, neben den bereits erwähnten Gründen, Typanalysen zu definieren ist gewesen, daß alle anderen Datenflußanalyseprobleme von einem Kontrollflußgraphen profitieren, der mit Typinformationen ausgestattet ist. Durch ihn sind andere Analysen weitaus genauer durchführbar, da die Menge der in Frage kommenden Programmteile durch Typinformationen relativ exakt bestimmbar ist. Daher empfiehlt es sich, zunächst immer eine Typanalyse durchzuführen.

Über diese Arbeit hinaus, sind folgende Frage- und Problemstellungen interes-

sant:

- Definition und Implementierung anderer Datenflußanalyseprobleme: code motion, very busy expressions, ... können ohne großen Aufwand übertragen werden.

- Nutzung der Typanalyse in einem Compiler für SOL : Die momentane Implementierung von SOL [HW97] nutzt Typinformationen noch nicht aus.

- Typanalyse für getypte Sprachen wie z.B. C++ oder Java: Der vorgestellte Datenflußanalyse-Rahmen ist auf getypte Sprachen übertragbar. Es stellt sich die Frage nach dem Nutzen einer solchen Analyse in hybriden Sprachen wie C++, in der nicht-objekt–orientierte Konstrukte vorherrschen.

- Weiterentwicklung von SOLAT: Die graphische Benutzerschnittstelle kann in einigen Punkten noch weiter verfeinert werden, wie z.B. komfortableres Suchen nach bestimmten Programmpunkten.

Literaturverzeichnis

[Ag94] Ageson, O. Constraint–Based Type Inference and Parametric Polymorphism, In *Proceedings of the 1st International Static Analysis Symposium (SAS'94)*, Namur, Belgium, Springer–Verlag, LNCS 864 (1994), 70–100.

[Ag95a] Ageson, O. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP'95)*, Arhus, Denmark, Springer–Verlag, LNCS 952 (1995), 2–26.

[Ag95b] Concrete Type Inference: Delivering Object-Oriented Applications. Dissertation, Stanford University, (1995).

[AH95] Ageson, O., und Hölzle, U. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object–Oriented Languages. In *Proceedings of the 10th ACM SIGPLAN Annual Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, *ACM SIGPLAN Notices 30*, 10, (1995), 91–107.

[AWL94] Aiken, A., Wimmers, E.L., und Lakshman, T.K. Soft Typing with Conditional Types, In *Conference Record of the 21th International Symposium on Principles of Programming Languages (POPL'94)*, Portland Oregon, (1994), 163–173.

[Ba92] Barnard, A. From Types to Dataflow: Code Analysis for an Object–Oriented Language. Dissertation, University of Manchester, England, (1992).

[BI82] Borning, A. H., und Ingalls, D. H. H. A Type Declaration and Inference System for Smalltalk. In *Conference Record of the 9th International Symposium on Principles of Programming Languages (POPL'82)*, Albuquerque, New Mexico, (1982), 133–141.

[Ca88] A Semantics of Multiple Inheritance. *Information and Computation*, 76(2-3), (1988), 138–164.

[Ch92] Chambers, C. The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object–Oriented Programming Languages, Dissertation, Stanford University, 1992.

[CC77] Cousot, P., und Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4^{th} Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, (1977), 238–252.

[CC79] Cousot, P., und Cousot, R. Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6^{th} Annual ACM Symposium on Principles of Programming Languages (POPL'79)*, San Antonio, Texas, (1979), 269–282.

[CC96] Cousot, P. The Abstract Interpretation Perspective. *ACM Workshop on Strategic Directions in Computing Research*, (1996).

[CDG96a] Chambers, C., Dean, J., und Grove, D. Whole-Program Optimization of Object–Oriented Languages. Technical Report 96-06-02, Dept. of Computer Science and Engineering, University of Washington, (1996).

[CDG96b] Chambers, C., Dean, J., und Grove, D. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report 96-11-02, Dept. of Computer Science and Engineering, University of Washington, (1996).

[CSH95] Carini, P.R., Srinivasan, H., und Hind, M. Flow–Sensitive Type Analysis for C++, IBM Research Report RC20267 Computer Science, (1995).

[DDGLC96] Dean, J., DeFouw, G., Grove, D., Litvinov, V., und Chambers, C. Vortex: An Optimizing Compiler for Object–Oriented Languages. In *Proceedings of the 11^{th} ACM SIGPLAN Annual Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, *ACM SIGPLAN Notices 31*, 10, (1996).

[DGC95] Dean, J., Grove, D., und Chambers, C. Optimization of Object–Oriented Programs using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP'95)*, Aarhus, Denmark, Springer–Verlag, LNCS 952 (1995), 77–101.

[DS84] Deutsch, L.P., und Schiffman, A.M. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the 11^{th} Annual ACM Symposium on Principles of Programming Languages (POPL'84)*, Salt Lake City, Utah, (1984), 297–302.

[Go95] Golubski, W. Type Analysis in Object–Oriented Languages — The Crucial Points. In *Proceedings of the 10th International Symposium on Computer and Information Sciences (ISCIS X)*, Izmir, Turkey, Vol. 2, (1995), 659–665.

[Go96] Golubski, W. Can Type Analysis by Abstract Interpretation be Practical? University of Siegen, Germany, Technical Report Informatik-Berichte, No. 96-06, (1996).

[Go97] Golubski, W. On the Trade-Off Between Precise and Efficient Type Analysis in Object-Oriented Programs. University of Siegen, Germany, Technical Report Informatik-Berichte, No. 97-01, (1997).

[Gr89] Graver, J. O. Type-Checking and Type-Inference for Objetc–Oriented Programming Languages. Dissertation, University of Illinois at Urbana–Champaign, (1989).

[Gr95] Grove, D. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings of CASCON'95 Centre for Advanced Studies Conference*, Toronto, Canada, (1995), 195–203.

[GJ90] Graver, J. O., und Johnson, R. E. A Type System for Smalltalk. In *Conference Record of the 17th International Symposium on Principles of Programming Languages (POPL'90)*, San Francisco, California, 1990, 136–150.

[GP97b] Golubski, W., und Pohlers, B. SOLAT- A Simple Object-Oriented Analyzing Tool. University of Siegen, Germany, Technical Report Informatik-Berichte, No. 97-02, (1997).

[GR83] Goldberg, A., und Robson, D. Smalltalk-80: The Language and Its Implementation. Addision-Wesley, (1983).

[He77] Hecht, M.S. Flow Analysis of Computer Programs. Elsevier North-Holland, N.J., USA, (1997).

[He93] Hense, A.V., und Smolka, G. Principal Types for Object-Oriented Languages. Technischer Bericht Nr. A 02/93 der Universität des Saarlandes, (1993).

[Hi69] Hindley, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the Mathematical Society*, 146, (1969), 29–60.

[Hö95] Hölzle, U. Adaptive Optimization in Self: Reconciling High Performance with Exporatory Programming, Dissertation, Stanford University, (1995).

[HCU91] Hölzle, U., Chambers, C., und Ungar, D. Optimizing Dynamically–Typed Object–Oriented Languages with Inline Caches. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP'91)*, Geneva, Switzerland, Springer–Verlag, LNCS 512 (1991), 21–38.

[HS77] Hunt, J.W., und Szymanski, A. Fast Algorithm for Computing Longest Common Subsequences, *Communications of the ACM*, 20(5), (1977), 350–353.

[HU94] Hölzle, U., und Ungar, D. Optimizing Dynamically–Dispatched Calls with Run–Time Feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 326–336.

[HW97] Heuser, H., und Wetzig, J. Entwurf und Realisierung einer Implementierung für eine ungetypte objekt–orientierte Programmiersprache. Universität-GH Siegen, Diplomarbeit, (1997).

[Jo86] Johnson, R.E. Type-Checking Smalltalk. In *Proceedings of the ACM SIGPLAN Annual Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, (1986), 315–321.

[JGZ88] Johnson, R.E., Graver, J.O., und Zurawski, L.W. TS: An Optimizing Compiler for Smalltalk. In *Proceedings of the ACM SIGPLAN Annual Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, (1988), 18–26.

[Kn93] Knoop, J. Optimal Interprocedural Program Optimization: A new Framework and its Application. Dissertation, University of Kiel, Germany, (1993).

[KG96a] Knoop, J., und Golubski, W. Abstract Interpretation: A Uniform Approach for Powerful Type Analysis and Classical Optimization of Object–Oriented Programs. In *Proceedings of the International Workshop "The White OO Nights" (WOON'96) (St. Petersburg, Russia)*, (1996), wird noch erscheinen.

[KG96b] Knoop, J., und Golubski, W. Precise Type Analysis by Abstract Interpretation. In: K. D. Reinartz (Ed.) PARS-Workshop 96, Mitteilungen - GI - Parallel-Algorithmen und Rechnernstrukturen, (1996), 59–68.

[KG96c] Knoop, J., und Golubski, W. Precise Type Analysis by Abstract Interpretation. University of Passau, Germany, Technical Report, Dept. of Mathematics and Computer Science, Technical Report,No. MIP-9619, (1996).

[KRS92] Knoop, J., Rüthing, O., und Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 224–234.

[KRS94a] Knoop, J., Rüthing, O., und Steffen, B. Optimal Code Motion: Theory and Practice. *Transactions on Programming Languages and Systems 16*, 4 (1994), 1117–1155.

[KRS94b] Knoop, J., Rüthing, O., und Steffen, B. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 147–158.

[KRS94c] Knoop, J., Rüthing, O., und Steffen, B. A Tool Kit for Constructing Optimal Interprocedural Data Flow Analyses. Fakultät für Mathematik und Informatik, Universität Passau, Germany, MIP–Bericht Nr. 9413, (1994).

[KRS95] Knoop, J., Rüthing, O., und Steffen, B. The Power of Assignment Motion. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, La Jolla, California, *SIGPLAN Notices 30*, 6 (1995), 233–245.

[KS92] Knoop, J., und Steffen, B. The Interprocedural Coincidence Theorem. In *Proceedings of the 4^{th} International Conference on Compiler Construction (CC'92)*, Paderborn, Germany, Springer–Verlag, LNCS 641 (1992), 125–140.

[KU77] Kam, J. B., und Ullman, J. D. Monotone Data Flow Analysis Frameworks. *Acta Informatica 7*, (1977), 309–317.

[Ma93] Marriot, K. Frameworks for Abstract Interpretation. *Acta Informatica 30*, (1993), 103–129.

[Mc96] McLennan, M.J. Object-Oriented Programming with [incr Tcl]/Building Mega-Widgets with [incr Tk]. AT&T Bell Laboratories, 1996.

[Mi78] Milner, R.A. A Theory of Type Polymorphism. *Journal of Computer and System Sciences*, 17, (1979), 348–375.

[MJ81] Muchnick, S. S., und Jones, N. D. (Eds.). Program Flow Analysis: Theory and Applications. Prentice Hall, Englewood Cliffs, New Jersey, (1981).

[MR79] Morel, E., und Renvoise, C. Global Optimization by Suppression of Partial Redundancies. *Commun. of the ACM 22*, 2 (1979), 96 — 103.

[OPS92] Oxhøj, N., Palsberg, J., und Schwartzbach, M. I. Making Type Inference Practical. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, Springer–Verlag, LNCS 615 (1992), 329 — 349.

[Pa96] Pande, H.D. Compile Time Analysis of C and C++ Systems. Dissertation, New Brunswick Rutgers University, New Jersey, (1996).

[PC94] Plevyak, J., und Chien, A.A. Precise Concrete Type Inference for Object–Oriented Languages. In *Proceedings of the 9th ACM SIGPLAN Annual Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, *ACM SIGPLAN Notices 29*, 10, (1994), 324 — 340.

[PC95a] Plevyak, J., und Chien, A.A. Type Directed Cloning for Object-Oriented Programs. In *Proccedings of the Workshop for Languages and Compilers for Parallel Computers*, Columbus, Ohio, (1995).

[PC95b] Plevyak, J., Zhang, X., und Chien, A.A. Obtaining Sequential Efficiency in Concurrent Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Principales of Programming Languages (POPL'95)*, San Francisco, California, (1995), 311–321.

[PC97] Plevyak, J., und Chien, A.A. Iterative Flow Analysis. Unveröffentliches Manuskript. Erhältlich über `http://www-csag.cs.uiuc.edu/`.

[PR94] Pande, H.D., und Ryder, B.G. Static Type Determination and Aliasing for C++. *Technical Report LCSR–TR–236, Rutgers University*, (1994).

[PS91] Palsberg, J., und Schwartzbach, M. I. Object–Oriented Type Inference. In *Proceedings of the 6th ACM SIGPLAN Annual Conference on Object–Oriented Programming*

Systems, Languages, and Applications (OOPSLA'91), *ACM SIGPLAN Notices 26*, 11 (1991), 146 — 161.

[PS93] Palsberg, J., und Schwartzbach, M. I. Object–Oriented Type Systems. John Wiley & Sons, (1993).

[PS94] Phillips, G., und Shepard, T. Static Typing Without Expilicit Types. Unveröffentlicher Bericht, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, (1994).

[Po97] Pohlers, B. Implementierung einer Typanalyse für ungetypte objekt–orientierte Sprachen. Universität Münster, Diplomarbeit, (1997).

[Pu97] Pundt, T. Dissertation (in Vorbereitung). Universität Münster, (1997).

[Sc86] Schmidt, D.A. Denotational Semantics - A Methodology for Language Development. Wm. C. Brown Publishers, Dubuque, Iowa, (1986).

[Sh91a] Shivers, O. Control-Flow Analysis of Higher-Order Languages. Dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, (1991).

[Sh91b] Shivers, O. Data-Flow Analysis and Type Recovery in Scheme. In P.Lee(Ed.) *Topics In Advanced Language Implementation*, MIT Press, Cambridge, (1991).

[St93] Steffen, B. Generating Data Flow Analysis Algorithms from Modal Specifications. *Science of Computer Programming 21*, (1993), 115–139.

[SP81] Sharir, M., und Pnueli, A. Two Approaches to Interprocedural Data Flow Analysis. In [MJ81], 1981, 189–233.

[Su81] Suzuki, N. Inferring Types in Smalltalk. In *Conference Record of the 8^{th} International Symposium on Principles of Programming Languages (POPL'81)*, Williamsburg, Virginia, 1981, 187–199.

[To90] Tofte, M. Type Inference for Polymorphic References. *Information and Computation*, 89(1), (1990), 1–34.

[US91] Ungar, D., und Smith, R.B. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, (1991).

[VHU92] Vitek, J., Horspool, R. N., und Uhl, J. S. Compile–Time Analysis of Object–Oriented Programs. In *Proceedings of the 4^{th} International Conference on Compiler*

Construction (CC'92), Paderborn, Germany, Springer–Verlag, LNCS 641 (1992), 236–250.

[WM95] Wilhelm, R., und Maurer, D. Compiler Design. Addison-Wesley Publishing Company, (1995).

[Zi82] Zima, H.P. Compilerbau I&II. Reihe Informatik: 36&37, Bibliographisches Institut, Mannheim, Wien, Zürich, (1982).