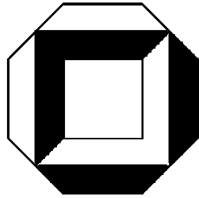


Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Dr. G. Goos

Studienarbeit

Ausnahmebehandlung in einem optimierenden Javaübersetzer

Till Riedel

Betreuender Mitarbeiter
Götz Lindenmaier

Verantwortlicher Betreuer
Prof. Dr. Gerhard Goos

Juni 2004

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Problemstellung . . . 3
- 1.2 Zielsetzung . . . 4
- 1.3 Aufbau der Arbeit . . . 4

2 Voraussetzungen **6**
- 2.1 Konzepte in Java . . . 6
  - 2.1.1 Geschützte Blöcke . . . 8
  - 2.1.2 Ausnahmeauslösende Javaoperationen . . . 8
  - 2.1.3 Klasseninstatiierung . . . 9
  - 2.1.4 Explizite Ausnahmen . . . 10
  - 2.1.5 Andere Ausnahmen . . . 10
- 2.2 Konzepte in Firm . . . 10
  - 2.2.1 Ausnahmebehandlung . . . 11
  - 2.2.2 Aufrufe . . . 12
  - 2.2.3 Ausnahmesemantik der Zwischensprachenoperationen . 12
- 2.3 Konzepte der Zielarchitektur . . . 14

3 Aufbau einer abstrakten Darstellung **15**
- 3.1 Abbildung von Java Konzepten auf Zwischensprachenebene . . 15
  - 3.1.1 Die Behandlungsroutinen . . . 15
  - 3.1.2 Der finally-Block . . . 17
  - 3.1.3 Ausnahmekontrollfluss . . . 17
- 3.2 Implementierung . . . 18
  - 3.2.1 Ausnahmekontrollfluss in geschützten Blöcken . . . 19
  - 3.2.2 Ausnahmebehaftete Operationen . . . 20
  - 3.2.3 Abspalten der Ausnahmesemantik . . . 21

4 Abbildung der abstrakten Darstellung **23**
- 4.1 Konkretisierung der Darstellung . . . 23
  - 4.1.1 Konkretisierung des intraprozeduralen Kontrollflusses . 23
  - 4.1.2 Konkretisierung des interprozeduralen Kontrollflusses . 24
- 4.2 Abbildung auf Zielsprachenebene . . . 24
  - 4.2.1 Verzweigung des intraprozeduralen Kontrollflusses . . . 25
  - 4.2.2 Verzweigung des interprozeduralen Kontrollflusses . . . 26

5 Implementierung und Experimentelle Ergebnisse **27**
5.1 Javaspezifischer Implementierungsaufwand 27
5.1.1 Implementierungsaufwand Laufzeitsystem 28
5.2 Laufzeitmessungen 28
5.2.1 Übersetzerperformanz 29
5.2.2 Programmperformanz 30
5.2.3 Ergebnisse 31

6 Zusammenfassung **32**

Kapitel 1

Einleitung

Man muss nicht mehr Katastrophen aus der Raumfahrt bemühen, um die Wichtigkeit von Ausnahmebehandlung zu zeigen. Studien wie [RSK+00] zeigen, dass solche Mechanismen allgemeine Akzeptanz unter Javaprogrammierern gefunden haben. Gepaart mit objektorientierten Programmierparadigmen stellen Ausnahmen einen intuitiven Mechanismus bereit, Programmverhalten inklusive Fehlerzuständen lückenlos zu spezifizieren.

Vielleicht sind es auch gerade Sprachmerkmale wie die Ausnahmebehandlung, die dazu geführt haben, dass sich die Programmiersprache Java in kurzer Zeit in dieser Weise behaupten konnte. Java hat sich längst von einer Entwicklungsumgebung für kleine Webapplikationen zu einer umfassend eingesetzten Programmiersprache gewandelt. Mit dieser Entwicklung entstehen auch neue Anforderungen für Javaübersetzer. Die Ausführung von Javakompilaten auf virtuellen Kellermaschinen resultiert zwar in einem Höchstmaß an Flexibilität und Sicherheit, führt jedoch auch künstliche Grenzen für die Optimierbarkeit ein.

So scheint es also logische Konsequenz die klaren Vorgaben der Java-Spezifikation mit den Vorteilen einer klassischen retargierbaren Übersetzerarchitektur zu verbinden. Diese Arbeit widmet sich der Implementierung von Ausnahmen und Ausnahmebehandlung aufbauend auf der SSA-basierten Zwischensprache *Firm* (siehe [TLB99]). Auf der Zwischensprache implementierte Optimierungen erlauben es gezielt auf die Probleme der Übersetzung objektorientierter Konstrukte einzugehen (siehe [Tra99]).

1.1 Problemstellung

Die Java Sprachspezifikation [*JLS*] enthält genaue Vorgaben der Ausnahmesemantik einzelner Konstrukte. Es wird jedoch schnell klar, dass die Umsetzung ein bedeutsamer Teil der Implementierung des gesamten Übersetzers ist. Die Gegenwart von Ausnahmen stellt hohe Anforderungen an die Korrektheit der Übersetzung und ist von kritischer Bedeutung für die Optimierung von Java.

Diese Arbeit soll zeigen, wie das durch die Sprachspezifikation vorgegebene Verhalten auf eine moderne Zwischensprache abgebildet werden kann. Die Zwischensprache *Firm* ist eine graphbasierte SSA-Darstellung, welche den Speicherzustand explizit modelliert. Cite Die Abbildung der Ausnahmebehandlung sollte die durch *Firm* vorgegebenen Abstrahierungen nutzen, so dass die Verwendung bereits implementierter Übersetzerphasen nicht eingeschränkt wird. Die Korrektheit von Optimierungen bezüglich der Zwischensprachsemantik muss ihre Anwendbarkeit auf Java sicherstellen. Die Javasemantik muss dazu lückenlos auf die Zwischensprache abgebildet werden.

In einem zweiten Schritt sollen dann die so erzeugten Zwischensprachkonstrukte auf eine Zielmaschine abgebildet werden. Die Unterstüzung für Ausnahmebehandlung variiert dabei abhängig von Betriebssystem und Prozessor. Für zielmaschinennahe Optimierungen und die Kodeselektion muss die Darstellung dazu entsprechend transformiert werden.

1.2 Zielsetzung

Bei der Lösung der allgemeinen Problemstellung soll diese Arbeit sich an einigen sinnvoll erscheinenden Prinzipien orientieren, welche sich aus den Entwurfsprinzipien der verwendeten Komponenten ableiten lassen. Java spezifiziert Ausnahmen als Folgen von Fehlern (siehe [*JLS*] §14). Die Effizienz der Ausnahmebehandlung ist also kein primäres Optimierungsziel. Ihre Darstellung sollte vielmehr die Übersetzung des Teils des Kontrollflusses ohne Ausnahmen so wenig wie möglich beeinflussen. Dies soll sowohl für den Übersetzungsaufwand wie auch für den zusätzlichen Aufwand zur Programmlaufzeit gelten. Um die Modularität und Retargierbarkeit des gesamten Systems nicht einzuschränken, sollten konkrete Entscheidungen über die Abbildung auf ein Zielsystem spät - möglichst erst bei der Kodeselektion - in der Übersetzerstruktur getroffen werden können.

Die Beschreibung beschränkt sich im Weiteren auf synchrone Sprachkonstrukte. Fehler in der Virtuellen Maschine oder im Betriebssystem werden hier nicht behandelt.

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Konzepte der Quellsprache Java und der Zwischensprache *Firm* in Bezug auf Ausnahmen und Ausnahmebehandlung vorgestellt. Abschnitt 2.3 schließt das Kapitel mit einigen Annahmen über ein mögliches Zielsystem.

Der Aufbau der Ausnahmedarstellung wird in Kapitel 3 besprochen. Im ersten Teil des Kapitels werden Abbildungsregeln geschaffen und Konstrukte eingeführt, die zur Abbildung benötigt werden. In Abschnitt 3.2 folgen dann konkrete Aspekte der Umsetzung von Teilen der Quellsprache.

Das darauf folgende Kapitel überspringt die Phase der Optimierungen und widmet sich der Abbildung auf ein Zielsystem. Dabei wird in zwei Schritten vorgegangen. Im ersten Schritt (Abschnitt 4.1) wird die Zwischensprachendarstellung lediglich transformiert. Dabei werden abstrakte Darstellungen konkretisiert und an die Zielsprache angenähert. Der zweite Schritt (Abschnitt 4.2) geht dann auf die Kodeselektion und die notwendige Unterstützung durch ein Laufzeitsystem ein.

Die eigentliche Implementierung innerhalb einer Übersetzerumgebung wird in Kapitel 5 vorgestellt. Messergebnisse von Übersetzerläufen sollen eine Relation zwischen Entwurfsentscheidungen und den Anforderungen realer Testprogramme herstellen.

Kapitel 6 schließt diese Arbeit mit einer kurzen Zusammenfassung.

Kapitel 2

Voraussetzungen

2.1 Konzepte in Java

Die *Java Language Specification* [JLS] führt Ausnahmen als ein Mittel ein, Fehlersituationen und daraus entstehendes Programmverhalten genau zu spezifizieren. Bei Verletzung semantischer Beschränkungen soll es dem Programmierer möglich sein, Aufsetzpunkte zu definieren, an denen der Programmverlauf fortgesetzt wird. Damit distanziert sich Java von der Praxis Fehlerzustände in Rückgabewerten zu kodieren und schafft gleichzeitig auch Instrumente, um mögliche Fehler in die Schnittstellenbeschreibung aufzunehmen.
Die Konzepte, die der statischen Überprüfung der Semantik zugrundeliegen, also schon während der Übersetzung zur Fehlererkennung beitragen, sollen hier jedoch nicht weiter besprochen werden. Vielmehr sollen diejenigen Konstrukte behandelt werden, welche das Laufzeitverhalten im Ausnahmefall spezifizieren.

Ausnahmen

Ist von Ausnahmen im Javakontext die Rede, so kann man zwischen zwei Konzepten unterscheiden. Mit dem Ausnahmekontrollfluss wird der reguläre Kontrollfluss unterbrochen und eine Ausnahmebehandlung eingeleitet: eine Ausnahme wird ausgelöst. Dies entspricht dem Ausnahmebegriff der Systemarchitektur.

Um anschließend eine differenzierte Behandlung zu ermöglichen wird die Ursache durch ein Objekt, die Ausnahme, repräsentiert. Über die Klasse der Ausnahme lässt sich eine Behandlungsroutine auswählen. Ausnahmeklassen bilden Hierarchien, welche eine gemeinsame Behandlung unter einer Superklasse vereinigter Ausnahmentypen zulässt. Alle Ausnahmeklassen erben von der Klasse *CoreThrowable*. Ausnahmen (oder besser: Ausnahmeobjekte) können wie jedes andere Objekt Felder und Methoden beinhalten.

Präzision von Ausnahmen

Ausnahmen sind in [JLS] §11.3.1 als „präzise“ definiert. Das heißt, dass die Programmsemantik den für den Benutzer sichtbaren Zustand zum Zeitpunkt der

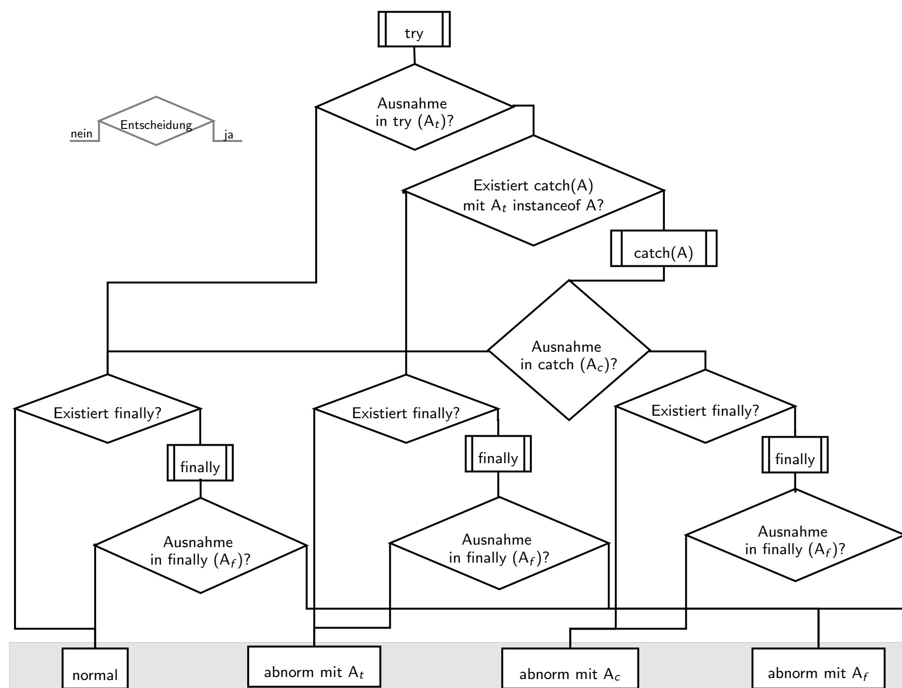


Abbildung 2.1: Ergebnis eines try-catch-finally Konstrukts nach [*JLS*] §14.19.2

Ausnahme genau beschreibt. Alle im Programm spezifizierten Effekte bis zur Ausnahme müssen bei ihrem Auftreten sichtbar sein. Genauso darf der Zustand auch nicht durch die Effekte nachfolgender Operationen verändert werden. Das heißt implizit natürlich auch, dass Ausnahmen in der spezifizierten Reihenfolge und unbedingt auftreten müssen. Was den benutzersichtbaren Zustand ausmacht, hängt intuitiv stark von der Laufzeitumgebung ab. Innerhalb der *Java Language Specification* wird dies leider nicht weitergehend spezifiziert. Im Weiteren wird davon ausgegangen, dass alle Zuweisungen an lebendige Variablen [1] sowie Ausgabeoperationen diesen Zustand ausmachen.

Dynamische Ausnahmebehandlung

Ein weiteres wichtiges Merkmal von Ausnahmen ist, dass ihre Behandlung nicht lokal innerhalb der Methode geschehen muss. Die Behandlungsroutine umschließt die ausnahmeauslösende Operation dynamisch. Im Fall einer intraprozedural unbehandelten Ausnahme gilt jeweils die Stelle des Aufrufs als ausnahmeauslösend. Von ihr aus wird rekursiv weiter die erste umschließende Behandlungsroutine gesucht. Dies geschieht, bis die Ausnahme behandelt oder die äußerste Schachtel verlassen wird. Im letzteren Fall wird der Ausführungsfaden terminiert und dessen Ausführungsumgebung (*ThreadGroup*)

[1] Diese Definition schließt Zugriffe über Methoden eines Laufzeitsystems ein. Da solche Zugriffe vollständig spezifiziert sind und bei geschlossener Übersetzung zur Übersetzungszeit bekannt sein müssen, stellt dies im vorliegenden Fall auch kein generelles Hindernis für Optimierungen dar.

über die unbehandelte Ausnahme benachrichtigt.
Anschließend wird der Programmfluss nicht an der Stelle des Auftretens der Ausnahme, sondern mit dem sich der Fehlerbehandlung anschließenden Kode fortgesetzt. Dadurch bleiben die Prinzipien der perfekten Schachtelung auf allen Ebenen gewahrt.

2.1.1 Geschützte Blöcke

Bereiche mit im Quelltext spezifizierter Ausnahmebehandlung werden in geschützten Blöcken zusammengefasst. In Java folgen diese Blöcke dem Schlüsselwort try. Sie werden von catch-Blöcken und einem optionalen finally-Block gefolgt, in denen die Fehlerbehandlung beschrieben ist.

catch-Blöcke

Das catch-Konstrukt definiert im Argument eine blocklokale Variable, die die aktuelle Ausnahme repräsentiert. Ist der Typ der so definierten Variablen eine Superklasse der ausgelösten Ausnahme, so kann diese von dem folgenden Block behandelt werden. Tritt eine Ausnahme bei Ausführung des geschützten Blocks auf, wird sie maximal von einem folgenden CATCH behandelt. Das im Quellcode textuell erste CATCH-Konstrukt, welches die Ausnahme behandeln kann, behandelt diese.

Behandelt keines der catch-Konstrukte die Ausnahme wird der try-Block abnorm mit dieser Ausnahme beendet. Wird die Ausnahme behandelt, so ist das Ergebnis des try-Blocks das des catch-Blocks (siehe auch Abschnitt 2.1.1 und Abbildung 2.1). Das Konstrukt als Ganzes verhält sich somit wieder wie eine einzelne potentiell ausnahmeauslösende Operation. Im Falle eines abnormen Endes wird also wiederum die dynamisch umschließende Behandlungsroutine ausgeführt.

finally-Blöcke

Zusätzlich kann ein finally-Block angegeben werden, der in jedem Fall nach Verlassen des geschützten Blocks ausgeführt werden muss (siehe [*JLS*] §14.19.2). Insbesondere gilt dies auch, wenn der Block mit einer Ausnahme, über einen Sprung (break [*JLS*] §14.14, continue [*JLS*] §14.15) oder durch normale Methodenrückkehr (return [*JLS*] §14.16) verlassen wird. Das Ergebnis eines try-catch-finally-Konstrukts insgesamt ist in Abbildung 2.1 dargestellt.

2.1.2 Ausnahmeauslösende Javaoperationen

Der Ausnahmemechanismus wird von Java dazu verwendet, Verletzung der Javasemantik zur Laufzeit zu signalisieren. Für Operationen, die abhängig von der Eingabe kein sinnvoll definiertes Ergebnis liefern können, schreibt die Java Spezifikation auszulösende Ausnahmen vor. Zusätzlich zu klassischen Ausnahmezuständen, wie die Division durch Null oder illegale Speicherzugriffe, werden auch statisch unentscheidbare Typkonflikte abgefangen und Typsicherheit zur Laufzeit sichergestellt.

Arithmetische Operationen

Die Division und der Teilerrest auf Ganzzahlen sind auf den meisten Prozessoren (wie z.B. denen der IA32-Architektur) keine reinen Datenflussoperationen, da Berechnungen mit einer Null als Divisor kein definiertes Ergebnis liefern. Stattdessen haben sie in diesem Fall einem Ausnahmezustand des Prozessors zur Folge. In Java löst die Operation in diesem Fall eine *ArithmeticException* aus (siehe [*JLS*] §15.17.2, bzw. §15.17.3). Damit sind diese beiden Operationen die einzigen arithmetischen Operationen, welche in Java eine Ausnahme auslösen können. Fehler, die durch Überlauf entstehen werden nicht auf diese Weise abgefangen (siehe [*JLS*] §4.2.2). Genauso lösen Fließkommaoperationen nie eine Ausnahme aus, da dort der der Java-Arithmetik zugrundeliegende IEEE Standard 754 immer ein definiertes Ergebnis vorsieht.

Qualifizierter Zugriff

Ein Methodenaufruf oder Feldausdruck löst im Fall, dass der qualifizierende Ausdruck [2] zu null ausgewertet wird, eine *NullPointerException* aus.

Typumwandlung

Wird im Programm eine explizite Typumwandlung vorgenommen, so muss das umzuwandelnde Objekt an den Zieltypen der Typumwandlung zuweisbar (siehe [*JLS*] §5.2) sein. Ist dies nicht der Fall, muss zur Laufzeit eine *ClassCastException* ausgelöst werden. Die Umwandlung von null ist immer erlaubt.

Reihungen

Ist die Reihung das null-Objekt und soll auf ein Element zugegriffen werden, so ist eine *NullPointerException* auszulösen (siehe [*JLS*] §15.26.1).

Wird zur Laufzeit versucht auf ein Element außerhalb von Reihungsgrenzen zuzugreifen, so wird in Java eine *ArrayIndexOutOfBoundsException* ausgelöst (siehe [*JLS*] §10.4).

Ähnlich wie bei einer Typumwandlung kann es notwendig sein, die Zuweisbarkeit auf Reihungen zur Laufzeit zu prüfen. Bei Reihungen von Referenztypen muss geprüft werden, ob die rechte Seite der Zuweisung auf den Elementtyp des Reihungsausdrucks zuweisbar ist. Ist dies nicht der Fall, muss eine *ArrayStoreException* ausgelöst werden (siehe [*JLS*] §10.10, §15.26.1).

2.1.3 Klasseninstatiierung

Allokation

Ein *OutOfMemoryError*, also die Nichtverfügbarkeit von Speicherressourcen, stellt konzeptionell einen Fehler der virtuellen Maschine dar und kann jederzeit ausgelöst werden. Obwohl es in der Praxis schwierig sein wird, können

[2] Ausdruck, welcher zu dem Objekt ausgewertet wird, über das auf Feld oder Methode zugegriffen wird. Typischerweise eine Variable, kann aber auch Resultat der Auswertung eines anderen Ausdrucks sein.

Javaprogramme versuchen solche Situationen zu behandeln. Es widerspricht der Spezifikation jedoch nicht, wenn diese lediglich an vorgegeben Stellen wie beim Versuch der Klasseninstatiierung oder bei Methodenaufrufen auftreten (siehe [*JLS*] §11.3.2). Die geforderte Präzision bezüglich des beobachteten Programmzustands muss jedoch gewährleistet sein.

Konstruktoren und Instanzinitialisierer

Unbehandelte Ausnahmen in Instanzinitialisierern und Konstruktoren treten an der Stelle des Instantiierungsausdrucks auf (siehe [*JLS*] §11.3).

Statische Initialisierer

Java schreibt vor, dass Ausnahmen in statischen Initialisierern bei der ersten Benutzung der Klasse ausgelöst werden (siehe [*JLS*] §11.3). Diese Forderung scheint innerhalb der vorliegenden Umgebung schwer und nur unter erhöhtem Aufwand zur Laufzeit umsetzbar zu sein. Daher wird diese Forderung hier im Folgenden nicht weiter beachtet. Stattdessen sollen solche Ausnahmen vor Ausführen der *main*-Methode auftreten.

Die Sprachspezifikation fordert weiter, dass alle Ausnahmen, welche nicht von der Klasse *Error* erben und innerhalb eines statischen Initialisierers auftreten, bei der ersten Benutzung einen *ExceptionInInitializerError* zur Folge haben. Dieses Ausnahmeobjekt kapselt die ursprüngliche Ausnahme (siehe [*JLS*] §14.17).

2.1.4 Explizite Ausnahmen

Zusätzlich zu den aus der Java-Semantik resultierenden Ausnahmesituationen, kann der Programmierer über eine throw-Anweisung Ausnahmen erzeugen. Die auszulösende Ausnahme ist Argument der Anweisung. Im Fall einer Auswertung zu null schreibt die Spezifikation eine *NullPointerException* vor.

2.1.5 Andere Ausnahmen

Wie in Abschnitt 1.2 bereits erwähnt, ist es nicht Ziel dieser Arbeit, asynchrone Ausnahmen oder Ausnahmen, die durch dynamisches Laden entstehen können, zu behandeln (obwohl diese Bestandteil der [*JLS*] §11.3.2 bzw. §11.5.1). Insbesondere gilt dies für Ausnahmenverhalten, welches aus der Spezifikation der *Java Virtual Machine* (siehe [*JLS*] §11.5.2, [*JVMS*]) abgeleitet werden kann.

2.2 Konzepte in Firm

Die Zwischensprache muss alle Restriktionen, die die Korrektheit der Optimierung gewährleisten, ausdrücken können. Die Anforderungen variieren dabei stark bezüglich der vorgegebenen Spezifikation. Um aggressiv in Hinblick auf Programmlaufzeit optimieren zu können, ist es dabei wichtig diese Restriktionen möglichst minimal zu halten, das heißt sie möglichst genau zu beschreiben, um so den Optimierungsraum zu vergrößern. Ist geringe Übersetzerlauf-

zeit jedoch ein weiteres Optimierungsziel, so sollte die Darstellung möglichst karg und allgemeingültig bleiben.

Firm stellt deshalb verschiedene Möglichkeiten bereit diese Restriktionen auszudrücken. So kann die Forderung der [*JLS*] nach Präzision (siehe Abschnitt 2.1) über Zustandsveränderungen modelliert werden. *Firm* unterscheidet zwei Arten von Zustand: Speicherzustand und Kontrollfluss.

Die Veränderung des Speicherzustands wird als Datenfluss abgebildet. Mit Hilfe von Φ-Operationen wird dessen Abhängigkeit vom Kontrollfluss dargestellt. *Firm* abstrahiert also Speicherbereiche als Variablen und Speicherabhängigkeiten als Definitions-Benutzungs-Beziehungen. Durch Beachten der Speicherabhängigkeiten können Operationen serialisiert werden und Operationen korrekt durch Optimierungen verschoben werden. Der Speicherzustand kann dabei disjunkt aufgeteilt werden, so dass diese Restriktionen beliebig genau modelliert werden können.

Ist keine Ausnahmebehandlung gefordert, reicht eine korrekte Serialisierung ausnahmebehafteter Operationen über diese Mechanismen aus. Wie die Anordnung von Lade- und Speicheroperationen über Speicherabhängigkeiten sichergestellt wird, so werden nun künstliche Speicherabhängigkeiten eingeführt, um andere Ausführungsreihenfolgen sicherzustellen. Dazu wird die Semantik der zu serialisierenden Operationen um Zugriffe auf abstrakte Ausnahmevariablen erweitert. Muss Operation A vor Operation B ausgeführt werden, so schreibt A konzeptuell auf diese Variable - während B sie liest. Eine Serialisierungskette kann erreicht werden, wenn Operationen gleichzeitig lesend und schreibend zugreifen. Sind verschiedene unabhängige Serialisierungsanforderungen gegeben, so können mehrere dieser abstrakten Variablen eingeführt werden. Operationen, für welche diese besonderen Speicherabhängigkeiten gelten, werden auch als fragil bezeichnet.

Um zu verhindern, dass durch Optimierungen Ausnahmen auf Pfaden auftreten, auf denen diese nicht aufgetreten sind, können zusätzlich Operationen an Grundblöcke gebunden werden. In *Firm* kann dazu ein Operatortyp mit *pinned* markiert werden.

2.2.1 Ausnahmebehandlung

Die Serialisierung über Speicherabhängigkeiten hat den Vorteil, dass sie keiner Modellierung durch den Kontrollfluss bedarf und so weniger zusätzlichen Aufwand erzeugt (siehe [Tra99]). Ist jedoch eine Ausnahmebehandlung wie im Falle von Java gefordert, so muss zusätzlich der Kontrollfluss modelliert werden, welcher aus dem Auftreten einer Ausnahme folgt. So stellen alle potentiell ausnahmeauslösenden Operationen bedingte Sprünge dar. Bedingung für den Sprung ist dabei der spezifizierte Zustand, der die Ausnahmesituation zur Folge hat. Da über den Speicherfluss bereits eine korrekte Serialisierung sichergestellt ist, müssen hierfür keine zusätzlichen Grundblöcke eingeführt werden. Die neu eingeführten Kontrollflussoperationen können zusammen mit dem

am Ende des Blocks stehenden Sprung zu erweiterten Grundblöcken zusammengefasst werden.

2.2.2 Aufrufe

Interprozedural gesehen können Aufrufe in Beginn und Rückkehr des Aufrufs aufgeteilt werden. Zur Beachtung von Ausnahmen reicht es für den Beginn aus, diesen über den Speicherzustand zu serialisieren. Bei der Rückkehr kann man zwischen der normalen und der abnormen Rückkehr unterscheiden. Für die normale Rückkehr müssen in Hinblick auf die Ausnahmebehandlung keine weiteren Schritte unternommen werden. Die abnorme Rückkehr sieht jedoch vor, dass die Ausnahmebehandlung an der Stelle des Aufrufs fortgesetzt wird.

In *Firm* wird die abnorme Rückkehr intraprozedural über eine Kontrollflusskante von der ausnahmeauslösenden Operation zum END-Knoten ausgedrückt. Der END-Knoten ist eindeutig für jede Methode und wird im Falle einer normalen Rückkehr über Kontrollflusskanten mit der Rückgabeoperation RETURN verbunden. Interprozudural wird der END-Knoten in die beiden Knoten ENDREG (für den normalen Kontrollfluss) und ENDEXCEPT (für den Ausnahmekontrollfluss) aufgeteilt.

Gibt man dem CALL-Knoten, welcher in *Firm* den Aufruf modelliert, eine Kontrollflusskante auf die aktuelle Behandlungsroutine, so wird sie interprozedural gesehen mit dem ENDEXCEPT-Knoten verbunden und der Ausnahmekontrollfluss wird darüber fortgesetzt. [3]

2.2.3 Ausnahmesemantik der Zwischensprachenoperationen

Welche Operationen unter welchen Bedingungen Ausnahmen auslösen können, ist sprachabhängig. *Firm* definiert einige seiner Operationen wie die Ganzahldivision (DIV) als ausnahmebehaftet (siehe Tabelle 2.1). Diese Operationen können einen Kontrollflussnachfolger für den Ausnahmefall haben, sind also wie schon angesprochen auch bedingte Sprünge. Ist kein Kontrollflussnachfolger mit einer solchen Operation verbunden, so wird angenommen, dass diese nie eine Ausnahme auslösen kann. Dies hat zur Konsequenz, dass alle potentiellen Ausnahmen explizit modelliert werden müssen, da ansonsten der Kontrollfluss unvollständig spezifiziert bleibt. Auch primär nicht operationsgebundene, asynchrone Ausnahmesituationen müssen immer über Operationen, die dann mit dem entsprechenden Ausnahmekontrollfluss verknüpft sind, im Kontrollfluss verankert werden.

Die Semantik der Divisionen (DIV, MOD, DIVMOD, QUOT) schreibt eine Ausnahme im Fall einer Null als Dividenden vor. Bei ungültigen Speicheradressen als Eingabe lösen Speicher- und Ladeoperationen Ausnahmen aus.

[3] *Firm* sieht bei Aufrufen (CALL-Knoten) abnorme Kanten für Kontrollfluss als auch für den Speicherzustand vor. Es ist an dieser Stelle jedoch nicht notwendig diese explizit zu behandeln. Beim Aufbau der Φ-Knoten für den Speicherzustand werden sie automatisch anstatt regulärer Kontrollflusskanten ausgewählt.

Operation	: Operanden	→ Resultate
RAISE	: $BB \times M \times P$	$\rightarrow X \times M$
CALL	: $BB \times M \times P \times data_1 \times \ldots \times data_n$	$\rightarrow M \times X \times data_{n+1} \times \ldots \times data_{n+m} \times M \times P$
QUOT	: $BB \times M \times float \times float$	$\rightarrow M \times X \times float$
DIVMOD	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int \times int$
DIV	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int$
MOD	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int$
LOAD	: $BB \times M \times P$	$\rightarrow M \times X \times data$
STORE	: $BB \times M \times P \times data$	$\rightarrow M \times X$
ALLOC	: $BB \times M \times int_u$	$\rightarrow M \times X \times P$

BB=Block P=Zeiger M=Speicher X=Kontrollfluss

Tabelle 2.1: Alle ausnahmeauslösenden Operationen in *Firm* (Quelle: [TLB99])

Operation	: Operanden	→ Resultate
CATCH	: $BB \times M$	$\rightarrow M \times P$
BOUNDS	: $BB \times M \times int_s \times int_s \times int_s$	$\rightarrow M \times X$
NEGSIZE	: $BB \times M \times int_s$	$\rightarrow M \times X$

BB=Block P=Zeiger M=Speicher X=Kontrollfluss

Tabelle 2.2: Zusätzlich eingeführte ausnahmeauslösende Operationen

Zu beachten ist, dass standardmäßig die Null als einzige ungültige Speicheradresse definiert ist. Auch beliebige implementierungsspezifische Fehler im Allokator können als Ausnahme abstrahiert werden. Zusätzliche Ausnahmesituationen können durch Einführen neuer ausnahmebehafteter Operationen oder bedingter Sprünge gefolgt von einer unbedingten Ausnahmeoperation (RAISE) ausgedrückt werden. Die unbedingte Ausnahmeoperation bietet die Möglichkeit einen Zeiger auf ein Ausnahmeobjekt anzuhängen, um so komplexere Ausnahmebehandlung zu ermöglichen. Ob und wie dieser Zeiger genutzt wird ist sprachabhängig.

Funktion	INT #	Auslösende Operation
Divide Error	0	`DIV IDIV`
Array Bounds Check	5	`BOUND`
Page Fault	14	Speicherzugriff oder Code Fetch

Tabelle 2.3: Ausschnitt der vom i386 unterstützen Ausnahmen (Quelle: [Int94])

2.3 Konzepte der Zielarchitektur

Es soll hier ein von der Zielarchitektur unabhängiges Vorgehen vorgestellt werden und doch sollen einige Konzepte einer möglichst repräsentativen Zielarchitektur angenommen werden. Üblicherweise unterstützen moderne Prozessoren verschiedene Ausnahmen (Signale), um Betriebssystemkonzepte umsetzen zu können. Folgt aus einer Ausnahme keine sinnvolle Aktion durch das Betriebssystem, wird das Betriebssystem versuchen das dem Prozess, welcher diese ausgelöst hat, zu signalisieren (siehe [Bar00]).

Posix-Signale setzen dieses Konzept um. Damit ein Prozess Signale erhalten kann, muss er eine Funktion registrieren, über die ihn das Betriebssystem zurückruft. Um eine Behandlung des Signals durchführen zu können, wird der Funktion der Prozessorkontext übergeben, den diese dann verändern kann, bevor versucht wird die Ausführung fortzusetzen.

Welche Arten von Ausnahmen unterstützt werden, hängt vom konkreten Prozessor und dem verwendeten Betriebssystem, bzw. der virtuellen Maschine ab, welche die Zielsprache interpretiert. Eine Ganzzahldivision durch Null löst meist eine Ausnahme aus (wobei u.U. wichtig ist, auf welchem Teil des Prozessors diese ausgeführt wird). Auch ist per Konvention die Adresse Null außerhalb des legal adressierbaren Bereichs. Welche anderen Adressen außerdem illegal sind, ist jedoch abhängig vom Betriebssystem. Andere Möglichkeiten Ausnahmesituationen zu überprüfen sind stark prozessorabhängig. Ein Beispiel ist der `BOUND`-Befehl auf Intel 386 basierenden Architekturen (siehe [Int94], [Int04]), welcher Reihungsgrenzen überprüft und bei Verletzung eine Ausnahme auslöst.

Kann keine spezielle Ausnahmeunterstützung angenommen werden, so kann der Ausnahmekontrollfluss immer auch in bedingte Sprünge umgesetzt werden (siehe Abschnitt 4.1.1).

Kapitel 3

Aufbau einer abstrakten Darstellung

Diese Kapitel beschreibt die Übersetzung der Javasemantik in die *Firm*-semantik. Es werden dazu zunächst allgemeine Paradigmen Umsetzung beschrieben, die dann in der in Abschnitt 3.2 beschriebenen Implementierung umgesetzt werden.

An einigen Stellen wird dabei *Firm* konzeptuell durch neue Konstrukte erweitert. Diese dienen vorallendingen der klaren Beschreibung und können in der Praxis wieder auf die bestehenden *Firm*-Schnittstellen zurückgeführt werden.

3.1 Abbildung von Java Konzepten auf Zwischensprachenebene

Ziel ist es den Ausnahmekontrollfluss möglichst effizient auf Zwischensprachenebene darzustellen. Die im Syntaxbaum vorhandene Semantik muss dabei vollständig in der Zwischensprache abgebildet werden. Zuerst wird die Abbildung der in Java vorgegebenen Blockstruktur auf *Firm* dargestellt. Danach wird beschrieben, wie das von Java im Ausnahmefall geforderte Verhalten sichergestellt wird.

3.1.1 Die Behandlungsroutinen

Alle zu einem geschützten Block gehörige `catch`-Blöcke werden unter einem gemeinsamen Einsprungspunkt zusammengefasst. Dort werden dann die verschiedenen Kontrollflüsse differenziert.

Über eine spezielle Ladeoperation (CATCH-Knoten) ohne konkreten Zeiger wird zunächst das aktuelle Ausnahmeobjekt bereitgestellt. Durch die Vermeidung einer konkreten Ladeoperation muss die Instantiierung eines konkreten Ausnahmeobjekts (noch) nicht dargestellt werden, solange sichergestellt wird, dass die Ladeoperation mit allen fragilen Operationen korrekt serialisiert wird

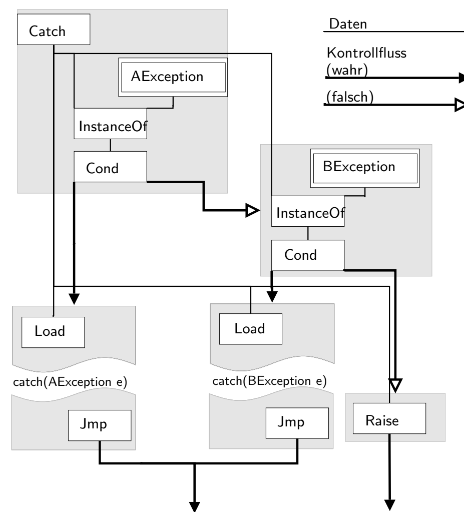


Abbildung 3.1: Zwei aufeinander folgende catch-Blöcke in *Firm*

(siehe 3.1.3). Die Allokation, Initialisierung und das Speichern des Ausnahmeobjekts kann so implizit nach dem Auslösen und vor dem Laden geschehen und muss nicht dargestellt werden. Dies ist ausreichend, da es keine weitere Möglichkeit gibt, auf die geladene Speicheradresse zuzugreifen. Wird diese Ladeoperation weiterhin als lesend und schreibend auf unbekannten Speicher angenommen, werden auch alle Seiteneffekte der Initialisierung des Ausnahmeobjekts abgedeckt. Eventuelle Ausnahmen während der Instantiierung können lediglich den Wert des geladenen Ausnahmeobjekts ändern.

Die Auswahlsemantik wird über geschachtelte bedingte Sprünge realisiert. Ein Instanztest auf den gerade geladenen Ausnahmeobjekt und dem Typ des catch-Arguments ist jeweils Bedingung und der catch-Block Ziel (siehe Abbildung 3.1). Die catch-Blöcke werden so in der durch den Quelltext vorgegeben Reihenfolge verkettet. Trifft keine der Bedingungen in der so entstandenen Kette zu, so wird an ihrem Ende die Ausnahme abermals über ein RAISE ausgelöst. Diese kann nun von äußeren Behandlungsroutinen behandelt werden.

Da die CATCH-Operation im Kopf der Kette einen Verweistyp liefert, der a priori (Instanztest) an das jeweilige blocklokale Ausnahmeobjekt zuweisbar ist, können alle Benutzungen des Ausnahmeobjekts als Kanten auf diesen Knoten dargestellt werden.

Der geschützte Block selber muss nicht explizit dargestellt werden. Fragile Operationen innerhalb des selben geschützten Blocks zeichnen sich lediglich dadurch aus, dass sie den selben Einsprungspunkt für den Ausnahmefall besitzen. Alle Operationen über die der geschützte Block oder die Behandlungsroutinen verlassen werden können, haben den mit finally markierten Block als direkten Kontrollflussnachfolger.

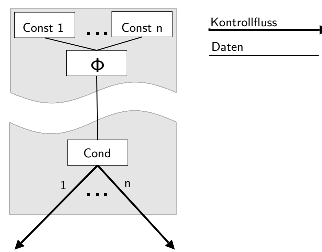


Abbildung 3.2: Kopf und Fuß des finally-Blocks in *Firm*

3.1.2 Der finally-Block

Eine Möglichkeit den finally-Block semantisch in der Zwischensprache zu erfassen, besteht darin, Duplikate des Blocks auf den betroffenen Pfaden einzufügen. Dies hätte jedoch offensichtlich zum Nachteil, dass redundanter Kode dargestellt würde.

Eine bessere Herangehensweise ist die Auswahl der Nachfolger anhand einer Variablen. Hierbei wird lediglich eine lokale Variable so definiert, dass sie einen eindeutigen Wert bezüglich des zu nehmenden Pfades besitzt. Alle aus dem Block austretenden Operationen können dann mit einem einzigen finally-Block verbunden werden. Der ursprüngliche Kontrollfluss wird über einen COND-Sprung mit Fallunterscheidung (switch) fortgesetzt. Das Sprungziel wird anhand der vorangegangenen Variablenbelegung gewählt.

Diese Darstellung hat den Vorteil, dass sie knapper und immer noch in die zuvor beschriebene Darstellung überführbar ist. Betrachtet man die resultierende Zwischensprachendarstellung, so wird hier mit Hilfe eine Φ-Knotens für eine Variable und einem COND-Knoten so etwas wie ein Φ-Knoten für den Kontrollflussnachfolger modelliert. Auf Hochsprachenebene würde das einem Speichern und Laden von Sprungadressen entsprechen. Da in *Firm* jedoch keine Kontrollflussobjekte existieren, wurden diese eindeutig auf Ganzzahlen abgebildet. Eine effiziente Kodeerzeugung im Sinne des eigentlichen Konstrukts bleibt weiterhin durch Erkennen des Musters möglich.

3.1.3 Ausnahmekontrollfluss

Die Umsetzung des von Java geforderten Verhaltens lässt sich in zwei Teilprobleme untergliedern. Das erste ist korrekte Serialisierung der Anweisungen. *Firm* stellt hierfür die in Abschnitt 2.2 beschriebenen Speicherabhängigkeiten bereit. Da in Java alle Ausnahmen gleichberechtigt sind und alle Befehle serialisiert werden, bedarf es nur einer abstrakten Variablen, um diese Abhängigkeit zu modellieren. Außerdem müssen alle Befehle mit Seiteneffekten abhängig von deren Zustand sein. Diese Tatsache erleichtert den Aufbau, da alle Befehle, welche den Speicherzustand verändern oder Ausnahmen auslösen, in der Reihenfolge ihres Auftretens über fortlaufende Speicherkanten verbunden werden können.

Das zweite Teilproblem ist die Darstellung der Bedingung unter der die Ausnahme auftritt. Für die Division ist die Semantik in *Firm* schon passend vorgegeben. Gleiches gilt trivialerweise für unbedingte Ausnahmen.

Bei einer zweiten Klasse von Operationen ist keine entsprechende *Firm*-Semantik vordefiniert oder eine 1:1 Übersetzung ist unglücklich, weil für den Aufbau des Ausnahmeobjekt notwendige Information verloren geht. So etwa bei den Lade- und Speicheroperationen, die eine Ausnahme bei unzulässigem Speicherzugriff auslösen. Die Information, ob die Ursache ein Zugriff über einen Nullzeiger oder eine Verletzung der Reihungsgrenzen war, ist jedoch an dieser Stelle schwer darzustellen. Weiterhin ist schwer zu testen, ob ein Zugriff überhaupt gültig ist. Es ist daher sinnvoll unverändert die Null als einzig ungültige Speicheradresse anzunehmen.

Eine Möglichkeit in diesem Fall ist eine Erweiterung des SELECT-Knoten, der dann ausnahmebehaftet wäre und zusätzlich Reihungsgrenzen übergeben bekommt. Dadurch wird jedoch dieser integrale Bestandteil der Zwischensprache abhängig von der Quellsprachensemantik, was die Wiederverwendbarkeit sprachunabhängiger Optimierungen unterminiert.

Die Lösung liegt hier in der Einführung neuer Operationen, welche im Normalfall Nulloperationen darstellen, jedoch unter bestimmten Bedingungen Ausnahmen auslösen können. Analog zu anderen Operationen kann man diese auch über den Speicherzustand serialisieren. Die den eigentlichen Fehlerzustand erzeugende Operation wird so geschützt und kann keine Ausnahme mehr auslösen.

Quellsprachunabhängige Optimierungen können diesen dann als Aufruf ohne Ergebnis betrachten, der abnorm zurückkehren kann. Ist genauere Information für aggressivere Optimierung notwendig, so kann die Semantik der Operation durch einen eigenen Graphen modelliert werden. Dieser kann genauere Analyseinformation bereitstellen. Die letztendliche Implementierung bleibt davon unberührt, sollte jedoch semantisch mit dem beschriebenen konform sein. Ist es für das Erreichen des Optimierungsziels nicht notwendig diese Information an allen Stellen verfügbar zu haben, so kann es ausreichen spezielle Optimierungen zu schreiben. Diese können die Semantik der neuen Operationen nutzen, und an geeigneter Stelle im Übersetzerlauf einzufügt werden.

3.2 Implementierung

Der von der Zerteilungsphase des Übersetzers aufgebaute Syntaxbaum wird in zwei Baumdurchläufen durch rekursiven Abstieg in die Zwischensprache *Firm* übersetzt.

Der erste Durchlauf dient lediglich dem Aufbau der Typhierarchie. Ausnahmetypen sind generell in den Java-Quellen des Laufzeitsystems beschrieben, so dass lediglich sichergestellt werden muss, dass die entsprechenden Quellen zerteilt wurden.

Die Darstellung des Ausnahmekontrollfluss und der Ausnahmeobjekte wird im zweiten Baumdurchlauf generiert. Hier werden die *Firm*-Graphen mit den Methodenkörpern aufgebaut.

3.2.1 Ausnahmekontrollfluss in geschützten Blöcken

Da try-Blöcke geschachtelt vorkommen können, reicht es während des Aufbaus nicht aus, den aktuellen zu betrachten. Es muss vielmehr sichergestellt werden, dass der äußere Kontext bei Verlassen des Blocks wiederhergestellt wird. Dazu wird ein Keller mit Kontexten angelegt. Der Kontext verweist auf den Einsprungspunkt für die zum try gehörigen Behandlungsroutinen, die wie in Abschnitt 3.1.1 beschrieben an diesen angehängt werden. Der Kontext wird vor Abstieg in einen geschützten Block eingekellert und anschließend wieder ausgekellert.

Unterstes Element ist der Kontext, welcher auf den Endblock zeigt. Für diese Ebene sind keine Behandlungsroutinen innerhalb der Methode vorhanden. Die korrespondierende Behandlungsroutine ist somit genau das Methodenende, so dass beim Auslösen einer Ausnahme eine abnorme Rückkehr eingeleitet wird (siehe Abschnitt 2.2.2). Beim Erstellen jeder ausnahmebehafteten Operation wird eine Kontrollflusskante zu dem Block erstellt, auf den das oberste Element des Kellers verweist (siehe Abschitt 3.2.2).

Einfügen des finally-Blocks

Analog zum Keller mit den Ausnahmebehandlungsroutinen wird ein Keller mit finally-Kontexten aufgebaut. Ein Kontext besteht aus einem Zeiger auf den Anfang des finally-Blocks, einer Liste, die jedem Vorgänger einen Nachfolger zuordnet und der Blockschachtelungstiefe in der sich das finally befindet.

Vor Erstellen des geschützten Blocks und der Behandlungsroutinen wird im Falle eines vorhandenen finally-Blocks ein neues Element auf den Keller gelegt. Als Anfangspunkt wird ein neuer Block erstellt, die Nachfolgerliste ist anfangs leer und die Schachtelungstiefe wird auf die aktuelle gesetzt.

An Stellen, an denen aus einem Block gesprungen werden kann, wird Sprungquelle und -ziel mit Hilfe des folgenden Algorithmus unter Kenntnis des aktuellen FINALLY-Kellers und der Schachtelungstiefe des Ziels verbunden:

- Ist der Keller leer oder das Sprungziel tiefer geschachtelt als das oberste Kellerelement, verbinde Sprungquelle und -ziel und kehre zurück.
- sonst:
  - Verbinde die Sprungquelle mit dem Anfangsblock des obersten Kellerelements.
  - Erstelle einen neuen Block. Trage den Block in die Nachfolgerliste des obersten Kellerelements ein.
  - Nehme das oberste Kellerelement von Keller. Verknüpfe einem Sprung aus dem neuerstellten Block als Quelle mit gleichbleibenden Sprungziel durch rekursive Anwendung des Algorithmus.

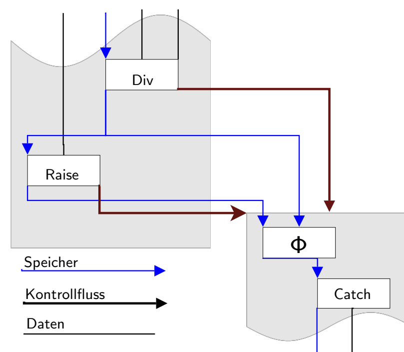


Abbildung 3.3: Serialisierung und Kontrollfluss zweier ausnahmebehafteter Operationen in *Firm*

Ist der Aufbau des geschützten Blocks und der Behandlungsroutinen abgeschlossen, enthält der oberste Kontext alle notwendigen Informationen um den aktuellen finally-Blocks wie in 3.1.2 beschrieben aufzubauen. Anschließend wird der Kontext ausgekellert.

3.2.2 Ausnahmebehaftete Operationen

Sind alle Behandlungsroutinen eines geschützten Blocks bekannt, dass heißt der zugehörige Einsprungspunkt für eine ausnahmebehaftete Operation ist oberstes Element des Kellers, kann man mit dem Abstieg in die Anweisungsblöcke beginnen. Anweisungen, welche eine Ausnahme auslösen können bekommen nun den aktuellen Enisprungpunkt für due Ausnhamenhandlung als Kontrollflussnachfolger im Ausnahmefall.

Explizite Ausnahmen

Bei Antreffen einer throw-Anweisung auszuführen sind lediglich das ausgewertete Argument mit einem neu angelegten RAISE-Knoten zu verknüpfen. Dieser wird widerum mit der aktuellen Behandlungsroutine verknüpft. Um zusätzlich sicherzustellen, dass, falls das Argument das Nullobjekt ist, auch eine *NullpointerException* erzeugt wird, wird zuvor eine eigentlich nicht benötigte Ladeoperation auf dem Objekt ausgeführt (siehe 3.2.3).

Aufrufe

Aufrufe benötigen lediglich eine Kontrollflusskante zur aktuellen Behandlungsroutine. Die Ausnahmebehandlung wird bei abnormer Rückkehr dort fortgesetzt (siehe Abschnitt 2.2.2).

Divisionen

Handelt es sich um eine Fließkommaoperation, so ist die Operation ausnahmefrei und ohne Speicherabhängigkeiten. Andernfalls ist wiederum eine Kontrollflusskante aufzubauen.

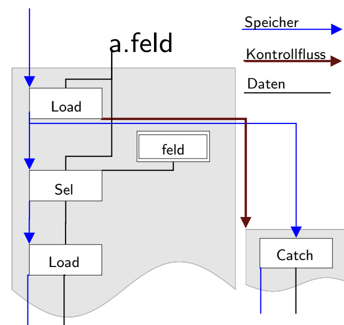


Abbildung 3.4: Potentielle *NullpointerException* bei Feldzugriff in *Firm*

Allokation

Es muss jede Allokation mit einer Kante zur aktuellen Behandlungsroutine aufgebaut werden, da entsprechende Fehler im Laufzeitsystem eine Ausnahmebehandlung innerhalb von Java zur Folge haben sollen (*OutOfMemoryError*). Ob es ausreicht Speichermangel an dieser Stelle abzufangen, hängt von der Laufzeitumgebung ab.

3.2.3 Abspalten der Ausnahmesemantik

Da manche Ausnahmesituationen sehr quellsprachenspezifisch sind, ist es nicht direkt möglich alle Operationen und ihr Ausnahmeverhalten als Einheit auf *Firm* abzubilden. Um nicht explizite Vergleiche in den Kontrollfluss einfügen zu müssen (siehe Abschnitt 2.2.1), werden diese Operationen zerlegt. Der reguläre Teil der Semantik wird über nicht ausnahmebehafte *Firm*-Operationen dargestellt. Die Ausnahmesemantik wird mit Hilfe zusätzlicher bedingter Sprungoperationen dargestellt. Im regulären Kontrollfluss haben diese keinen Effekt, sie definieren jedoch genau Position und Bedingung für das Eintreten in den entsprechenden Ausnahmekontrollfluss.

Feld- und Methodenzugriffe

Die in Java einzig möglichen illegalen Speicherzugriffe sind Reihungszugriffe mit ungültigem Index (siehe 3.2.3) und Zugriffe über einen Nullzeiger. Zum Abfangen eines Nullzeigerzugriffs ist es ausreichend einem qualifizierten Feld- oder Methodenzugriff einen Zugriff auf das Ergebnis des qualifizierenden Ausdrucks voranzustellen. Eine entsprechende LOAD-Operation löst genau dann eine Ausnahme aus, wenn hier auf den Nullzeiger zugegriffen wird. Ist das Objekt ungleich dem Nullzeiger, wird so immer das Resultat einer vorangegangenen Allokation, also eine legale Speicheradresse, geladen, hat also keinen Effekt.

Man fügt also einen zusätzlichen LOAD-Knoten mit dem ausgewerteten qualifizierenden Ausdruck als Eingang ein. Von diesem LOAD gehen außer dem Speicherzustand und einer eingefügten Kante zur aktuellen Behandlungsroutine keine weiteren Kanten aus. Das Ausnutzen dieses Teils der Semantik

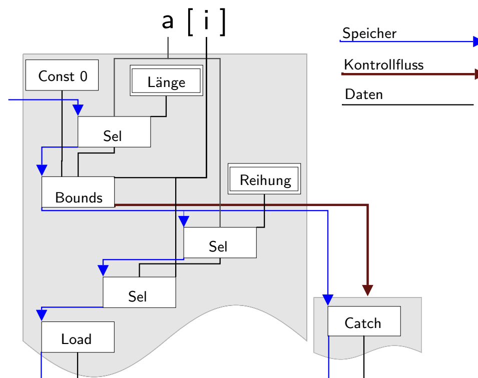


Abbildung 3.5: Reihungszugriff mit Ausnahmekontrollfluss in *Firm*

des LOAD-Knotens hat den Vorteil, dass auch sprachunabhängige Optimierungen und Analysen die vollständige Semantik der dargestellten Operation erfassen können.

Unqualifizierte Zugriffe sowie Zugriffe über this bzw. super können keine Ausnahmen auslösen, da ein gültiger Zeiger schon Voraussetzung für den Methodenaufruf ist.

Reihungszugriffe

Um der Semantik von Reihungszugriffen in Bezug auf Ausnahmen Rechnung zu tragen wird eine neue Zwischensprachen-Operation (BOUNDS-Knoten) eingeführt, die den Index und die Grenzen als Argument hat. Ist der Index innerhalb der Grenzen, so hat diese Operation keinen Effekt, sie liefert auch kein Ergebnis zurück. Andernfalls erzeugt sie eine *ArrayIndexOutOfBoundsException* und springt in die mit ihr zu verknüpfenden Behandlungsroutine. Die untere Grenze ist hier immer gleich Null, die obere Grenze wird aus der die Reihung umgebenden Struktur ausgelesen. Eingefügt wird diese Operation vor allen anderen Operationen, die für den Reihungszugriff notwendig sind. Die vorher abzufangende *NullPointerException* ergibt sich automatisch aus dem Zugriff auf das Feld, welches die obere Reihungsgrenze enthält, und wird standardkonform zuerst abgefangen. Für mehrdimensionale Reihungen werden BOUNDS-Knoten für jede Dimension einzeln angelegt.

Dynamische Reihungsallokation

NegativeArrayIndexExceptions können nicht erst bei der Allokation abgefangen werden, da die Darstellung der zu allozierenden Größe nicht vorzeichenbehaftet ist. Eine Allokation eigentlich negativer Größe wäre so durchaus semantisch korrekt. Es ist daher notwendig eine Operation voranzustellen, welche bei negativer Eingabe eine solche Ausnahme auslöst.

Kapitel 4

Abbildung der abstrakten Darstellung

4.1 Konkretisierung der Darstellung

Der CATCH-Knoten und die fragilen Operationen erlauben java-spezifische Teile der Ausnahmenimplementierung zu abstrahieren. So muss erst in einer späten Übersetzerphase festgelegt werden, wie das von Java vorgeschriebene Laufzeitverhalten erzeugt wird. Hat man ein komplexes Laufzeitsystem wie eine virtuelle Maschine, so kann die Erzeugung von Ausnahmeobjekten zum Beispiel komplett außerhalb des Prozedurkontexts geschehen. Im Folgenden wird eine typische Vorgehensweise vorgestellt diese Festlegung auf ein Zielsystem vorzubereiten. Ergebnis soll eine Darstellung sein, welche Ausgangspunkt für zielsystemnahe Optimierungen und die Kodeerzeugung ist.

4.1.1 Konkretisierung des intraprozeduralen Kontrollflusses

Wird wie hier beschrieben die Allokation komplett innerhalb des Programmkontexts vorgenommen, kann der CATCH-Knoten auf eine lokale Variable abgebildet werden. Im Falle des RAISE-Operation ist ihr Inhalt genau der Verweis auf das Ausnahmeobjekt im Argument der Operation. An dieser Stelle verliert dieser Eingang des RAISE-Knotens seine Bedeutung und die Operation an sich kann an dieser Stelle schon auf einen unbedingten Sprung abgebildet werden.

Für andere ausnahmeauslösende Operationen sind auf dem zur CATCH-Operation führenden Kontrollflusspfad Blöcke mit Allokatoren einzusetzen. Zusätzlich zur Allokation müssen genauso wie bei einer expliziten Allokation durch `new` eventuelle Initialisierer aufgerufen werden. Alle eingefügten Operationen sind korrekt mit den Nachfolgern des CATCH-Knotens über den Speicherzustand zu serialisieren.
Die Klasse des zu erzeugenden Objektes hängt von der auslösenden Operation ab (siehe 3.2.2).

Transformation des intraprozeduralen Kontrollflusses

Ist die Zielarchitektur ein konkreter Prozessor, so ist es unwahrscheinlich, dass alle ausnahmeauslösenden Operationen auf einen entsprechenden Befehl abgebildet werden können. Außerdem könnten zielsprachennahe Optimierungen und die Kodeerzeugung selber keine (volle) Unterstützung für einen Ausnahmekontrollfluss haben. In diesen Fällen macht es Sinn, den Ausnahmekontrollfluss teilweise oder vollständig auf herkömmlichen Kontrollfluss, also bedingte Sprünge, abzubilden.

Wie schon in Abschnitt 3.1.3 in Bezug auf Optimierungen angemerkt wurde, kann man die Semantik von ausnahmebehafteten Operation als eigene Graphen darstellen. Innerhalb der Graphen sind lediglich RAISE-Operationen erlaubt, welche problemlos auf Sprünge abzubilden sind (s.o). Diese Graphen kann man dann anstelle der Operation einsetzen. Ist im Übersetzer eine entsprechende Optimierung zum Einsetzen von Methodenaufrufen bereits implementiert, so kann unter Umständen diese genutzt werden.

4.1.2 Konkretisierung des interprozeduralen Kontrollflusses

In Abschnitt 4.1.1 wurde der Aufruf als ausnahmeauslösende Operation vernachlässigt. In diesem Fall wird das Ausnahmeobjekt vor Beendigung der Methode alloziert und initialisiert. Es muss also ein Mechanismus gefunden werden um das Ausnahmeobjekt zurückzugeben. Prinzipiell kann dies genauso wie bei einer Rückgabe über RETURN geschehen. Das zurückgegebene Objekt ist in diesem Fall das Argument der RAISE-Operation. Die lokale Variable, welche die CATCH-Operation abbildet, kann also in diesem Fall die Rückgabe des Aufrufs als Ergebnis bekommen. Jeder Methodentyp wird dazu um einen alternativen Rückgabetyp, einen Verweis auf ein Ausnahmeobjekt, erweitert.

Normalisierung abnormer Methodenenden

Um abnorme Rückgaben mit den Mitteln des regulären Kontrollflusses auszudrücken, müssen die abnormen Methodenenden in normale Rückkehroperationen umgesetzt werden. Zusätzlich ist es dann notwendig den Zustand der abnormen Rückkehr und die Rückgabe anderweitig zu kodieren. Das Ausnahmeobjekt wird dann an einer festgelegten Speicherstelle abgelegt. Ist bei einer Rückkehr dieses Objekt nicht das Nullobjekt, wird ein nach dem Aufruf eingefügter bedingter Sprung erreicht. Dieser führt zu einem Block, welcher eine lokale Variable auf das Objekt setzt und die Speicherstelle wieder auf das Nullobjekt setzt. Die Ausnahme kann dann lokal ausgelöst und der Ausnahmekontrollfluss fortgesetzt werden.

4.2 Abbildung auf Zielsprachenebene

An dieser Stelle haben nun alle noch vorhandenen fragilen Operationen Entsprechungen in der Zielsprache. Das hier angenommene Zielsystem setzt vom

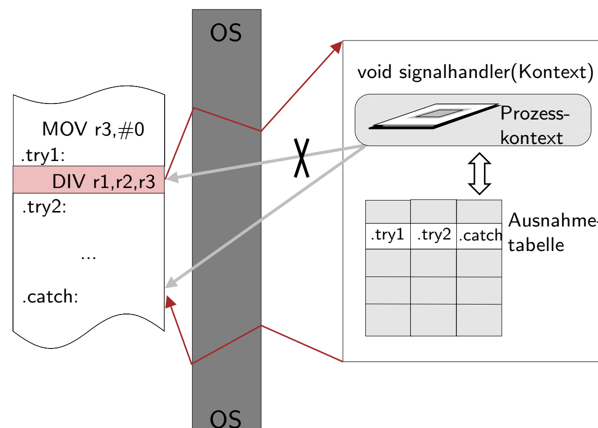


Abbildung 4.1: Ausnahmekontrollfluss über Signale und Ausnahmetabelle

Prozessor ausgelöste Ausnahmen, sofern sie nicht vom Betriebssystem behandelt werden können, in Posix-Signale um.

4.2.1 Verzweigung des intraprozeduralen Kontrollflusses

Ein Prozess kann diese Signale behandeln, indem er durch einen Systemaufruf eine Behandlungsroutine registriert. Die Behandlungsroutine entspricht jedoch nicht der Behandlungsroutine im zuvor gebrauchten Sinne. Eine direkte Abbildung hätte zur Folge, dass für jeden Wechsel der Behandlungsroutine ein teuerer Systemaufruf eingefügt werden müsste. Einem Prozessorzustand werden stattdessen über den Programmzeiger eindeutig Ausnahmekontrollflussnachfolger zugeordnet.

Ausnahmetabellen

Eine statische Zuordnung ist möglich, solange der Programmtext immer an eine feste Speicherstelle geladen wird. In einem Zielsystem mit virtueller Speicherverwaltung kann dies gewährleistet werden. Einzelne Punkte im Kontrollfluss und Sprungziele können in diesem Fall über absolute Bindermarken adressiert werden.

Die Ausnahmetabelle ordnet einem Abschnitt im Programmtext, gegeben durch Anfangs- und Endmarke, einen Einsprungspunkt im Ausnahmefall zu. Diese Tabelle wird in einem separaten Datensegment des Programms abgelegt, so dass diese auch bei getrennter Übersetzung erweitert werden kann. Durch Abbildung über die Ausnahmetabelle reicht es, eine einzige Ausnahmebehandlungsroutine zu registrieren, welche dann den aktuellen Programmkontext auf Kontrollfluss abbildet.

Kodeerzeugung für fragile Operationen

Kodeerzeugung für den Ausnahmefall kann genauso wie für anderen Kontrollfluss geschehen, außer dass statt eines Sprungbefehls ein Eintrag in die Ausnahmetabelle eingefügt wird. Voraussetzung ist, dass Marken vor und hinter dem Bereich eingefügt werden, innerhalb welchem Ausnahmen ausgelöst werden können. Diese und die Marke des Sprungziels bilden einen Tabelleneintrag.

Verändern des Prozedurkontexts

Vor der Ausführung des eigentlichen Programms wird durch das Laufzeitsystem eine Ausnahmebehandlungsroutine registriert. Kommt es zu einer Ausnahme während der Programmausführung ruft das Betriebssystem diese auf und übergibt ihr den Prozedurkontext zum Zeitpunkt der Ausnahme. Ohne weiteres Zutun würde dieser nach Austreten aus der Behandlungsroutine wiederhergestellt.

Bevor dies jedoch geschieht, kann die Behandlungsroutine nun den im Kontext abgelegten Programmzähler des Prozessors verändern. Dazu wird für den abgespeicherten Programmpunkt ein korrespondierender Eintrag in der Ausnahmetabelle gesucht. Anschließend wird der Programmzähler dann mit dem Sprungziel aus der Ausnahmetabelle ersetzt. Nach Wiederherstellen des Kontexts durch das Betriebssystem wird nun das Programm mit der zuvor verknüpften Ausnahmebehandlung fortgesetzt.

Explizite Ausnahmen brauchen diesen Mechanismus nicht, sie können durch unbedingte Sprünge abgebildet werden.

4.2.2 Verzweigung des interprozeduralen Kontrollflusses

Es ist sinnvoll, dass die aufgerufene Prozedur den Kontrollfluss des Aufrufers verändert. Da so kein zusätzlicher Laufzeitaufwand durch Kodierung und Prüfen des Zustands bei normaler Rückkehr entsteht.

Die Befehle zur abnormen Rückkehr können für jeden Endblock erzeugt werden. In der Zwischendarstellung endet der Kontrollfluss im Endblock. Im Gegensatz zur normalen Rückkehr wird dieser im abnormen Fall auch wirklich erreicht, da zuvor keine Rückkehroperation steht. Über dort platzierte Operationen kann also die Ausnahmebehandlung fortgesetzt werden.

Der weitere Kontrollfluss ist jedoch nicht statisch bekannt, sondern hängt vom Aufrufer ab und ist über diesen in der Ausnahmetabelle kodiert. Mit Hilfe der normalen Rückkehradresse kann die entsprechende abnorme Rückkehradresse ausgelesen werden. Wie bei einer normalen Rückkehr wird der Kontext des Aufrufers wiederhergestellt, jedoch mit geänderter Rückkehradresse. Wie das Ausnahmeobjekt übergeben wird, kann über eine individuelle Aufrufkonvention festgelegt werden.

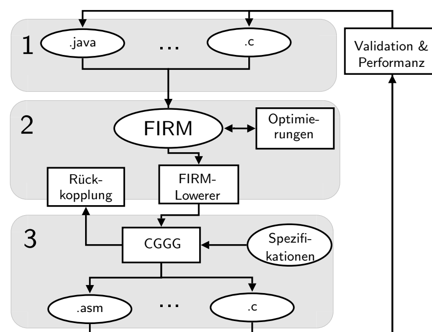
Kapitel 5

Implementierung und Experimentelle Ergebnisse

Basis für die Implementierung stellt die CRS Umgebung (siehe Abbildung 5.1) mit dem Jack-Frontend dar. Jack ist eine Modifikation des Javaübersetzers Jikes (siehe [IBM04]), welche die graphbasierte SSA-Zwischensprachendarstellung *Firm* aufbaut. Auf der Zwischensprache ist es möglich quellsprachen-unabhängige Optimierungen einzusetzen, die gemeinsam mit den benötigten Konstruktoren für die Graphen innerhalb einer Bibliothek bereitgestellt werden. Kodeerzeuger für unterschiedliche Zielarchitekturen stehen zur Verfügung.

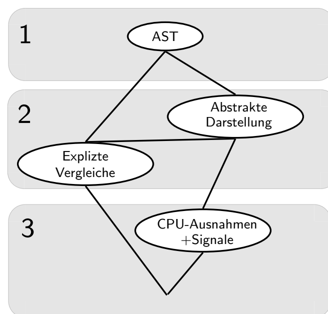
5.1 Javaspezifischer Implementierungsaufwand

Nach dem Zerteilen und der semantischen Analyse durch den Jikes Übersetzer werden durch Absteigen auf dem erzeugten Syntaxbaum *Firm*-Graphen aufgebaut. Die Prozedur, welche das try-catch-finally-Konstrukt aufbaut, ist in ca. 150 Zeilen C-Kode implementiert. Der Quellcode, der für die Verzweigung des Kontrollflusses an ausnahmenbehafteten Operationen zusätzlich notwen-



1: Frontend 2: Optimierungen 3: Backend

Abbildung 5.1: Die Übersetzerumgebung CRS



1: Frontend 2: Optimierungen 3: Backend

Abbildung 5.2: Darstellungsalternativen

dig ist, ist mit 1-10 Zeilen Quellcode pro Operationstyp sehr kompakt. Die Einführung zusätzlicher Operationen und ihrer Semantik bedurfte ca. 350 Zeilen Quellcode. Die Implementierung der in Abschnitt 4.1 beschriebenen Konkretisierungen umfasst ca. 600 Zeilen Quellcode. Die Implementierung aller javaspezifischen Teile ist mit unter 1500 Zeilen Quellcode gut überschaubar. Die zusätzliche Programmlogik wird dabei an wenigen klar definierten Stellen eingeführt. Algorithmisch sind dabei keine besonderen Hürden zu nehmen. Alles in allem kann so ein hohes Maß an Wartbarkeit und Stabilität erreicht werden.

5.1.1 Implementierungsaufwand Laufzeitsystem

Das Laufzeitsystem, welches die Verzweigung des Ausnahmekontrollflusses übernimmt, umfasst 165 Zeilen C-Quellcode. Die Implementierung ist dabei zielsystemspezifisch. Die vorliegende Implementierung wurde auf Intel Pentium 4 und AMD Athlon XP Prozessoren mit IA32-Architektur unter Linux (glibc 2.3) und FreeBSD (Version 5.1) getestet. Wird der Posix-Signalmechanismus auf einem alternativen Zielsystem bereitgestellt, sind lediglich Änderungen bezüglich des Zugriffs auf den Prozessorkontext notwendig.

Die Erzeugung der für das Laufzeitsystem notwendigen Tabellen durch den Kodegenerator ist durch die Verwendung von Bindermarken trivial. Es reicht das Einfügen der Marken in den erzeugten Kode und der Verweis auf sie in ein Segment, welches als Ausnahmetabelle dient. Werden die Marken der ausnahmeauslösenden Operation beim Erstellen immer hinten an die Tabelle angefügt, ist die Ausnahmetabelle außerdem geordnet und die Suche auf n Einträgen ist so zur Laufzeit auf maximal $log(n)$ Zugriffe beschränkt.

5.2 Laufzeitmessungen

Eine Breite Stichprobe und differenzierte Messergebnisse sind an dieser Stelle methodisch wünschenswert. Es wird jedoch auf diese sorgfältige Analyse auf Grund des Entwicklungsstandes des Gesamtsystems verzichtet. Um sichere

Übersetzerphase	Darstellung			
	Keine	Abstrakte		Explizite
Prozeduraufbau	6.9s	7.5s		7.8s
Transformationen und Analysen	19 s	40 s	(36 s)	60 s
Optimierungen I	3.4s	4.7s		13.7s
Interprozeduraler Darstellung	0.2s	0.3s		0.3s
Inlining und lokale Optimierungen	0.2s	0.3s		0.4s
Annähern an das Zielsystem	5.3s	5.9s		5.3s
Optimierungen II	10.6s	29.3s	(25.1s)	40.3s
Kodegenerator	51 s	68 s	(63 s)	71 s

Tabelle 5.1: Übersetzerlaufzeiten für HeapSort je nach Darstellung

Aussagen in Hinblick auf eine realistische Implementierung treffen zu können ist die Übersetzerumgebung an sich zu experimentell und unvollständig. Es werden stattdessen exemplarisch Messergebnisse vorgestellt, welche jedoch eine vorsichtige Bewertung der vorgestellten Implementierung zulassen.

Alle Zeitmessungen beziehen sich auf ein System mit 2,4 GHz getakteten Pentium4-Prozessor von Intel und ca. 500 Mb freiem Hauptspeicher. Als Betriebsystem wurde Linux mit der Kernversion 2.4 und der glibc-Version 2.3 verwendet.

5.2.1 Übersetzerperformanz

Tabelle 5.1 zeigt die Laufzeit der relevanten Phasen eines Übersetzerlaufs mit unterschiedlichem Grad der Unterstützung für Ausnahmen. Es wurde die HeapSort-Benchmark der JavaGrande Gruppe (siehe [HBS+99]) zusammen mit einem Javalaufzeitsystem in einem einzigen Übersetzerlauf übersetzt.

Die Spalte „Keine Darstellung" zeigt die Laufzeit ohne jegliche Ausnahmenunterstützung. catch-Blöcke sind hier unerreichbar und werden gar nicht erst aufgebaut. Die erzeugte Darstellung ist nicht korrekt falls eine Ausnahme auftreten kann. Sie ist nur als Vergleichspunkt zu sehen.

Die Spalte „Abstrakte Darstellung" bezieht sich auf die hier vorgestellte Implementierung. Aus dem Syntaxbaum wird eine abstrakte Darstellung ohne explizite Abfragen der Ausnahmebedingung und ohne Allokation der Ausnahmeobjekte. Es wird allerdings keine Ausnahmeunterstützung im Zielsystem angenommen und Ausnahmebedingungen werden vor der Kodegenerierung durch Vergleiche und bedingte Sprünge ersetzt. Die Werte in Klammern beziehen sich auf eine Darstellung, in der nach dem Annähern an das Zielsystem die Division durch Null und das Laden eines Nullzeigers auf dem Zielsystem genutzt werden.

Die Implementierung „Explizite Darstellung" baut die dagegen Vergleiche auf die Ausnahmebedingung und die Allokation der Ausnahmeobjekte direkt explizit auf.

Testprogramm	Unterstützung (Darstellung) Keine	Nur Jmp (Abstrakt)	 (Explizit)	Div / Load
HeapSortBench	3.4s	5.5s	5.5s	5.0s
Queens	37.5s	53.4	53.4	48.9s
BitSieve	13.8s	20.7s	21.1s	17.6s
Exceptions	-	3.8s	3.9s	8.6s + 11s

Tabelle 5.2: Laufzeitmessung der Testprogramme in Abhängigkeit von Laufzeitsystem-Unterstützung und Darstellung während der Übersetzung

Durch die abstrakte und damit knappe Darstellung der vorliegenden Implemnetierung, kann schon beim Aufbau der Aufwand leicht reduziert werden. Die Messungen zeigen insgesamt den erhöhten Übersetzungsaufwand durch die Präsenz von Ausnahmen. Die Laufzeit der Transformationen und Analysen auf *Firm* wird mehr als verdoppelt. Die Messergebnisse zeigen jedoch auch hier klar einen Vorteil der gewählten Darstellung des Ausnahmekontrollflusses. Gegenüber einer expliziten Darstellung profitieren alle Tranformationsphasen des Übersetzers von der knapperen Form der Darstellung („Abstrakte Darstellung“ gegenüber „Explizite Darstellung“) . Interessanterweise verkleinert sich auch der Aufwand für zielsystemnahe Optimierungen nach dem Aufbau der abstrakteren Darstellung, obwohl nach der Annäherungsphase der Abstraktionsgrad der Darstellungen wieder gleich ist. Die Umsetzung mancher Ausnahmesituation durch den Kodegenerator (Werte in Klammern) beschleunigt die zielsystemnahen Optimierungen um 15% und den Kodegenerator selber um mehr als 5%. Dies zeigt abermals, dass die Darstellung über zusätzliche Grundblöcke teuer ist.

Wichtiger als die Geschwindigkeit des Übersetzers ist die Geschwindigkeit des übersetzten Programms. Spalten 2-4 von Tabelle 5.2 entsprechen denen in Tabelle 5.1. Die letzte Spalte resultiert aus der Übersetzung auf ein System mit Laufzeitunterstützung für Division durch Null und Laden des Nullzeigers (Tabelle 5.1: Ergebnisse in Klammern). Die erste Zeile von Tabelle 5.2 zeigt die Ausführungszeit des Ergebnisses der Übersetzerläufe aus dem vorigen Abschnitt. Die Ergebnisse zeigen klar den Einfluss der Ausnahmebehandlung auf den regulären Kontrollfluss. Die ersten drei Programme lösen keine Ausnahmen aus, haben jedoch eine deutlich längere Laufzeit falls Ausnahmen in Betracht gezogen werden. Die Nutzung der Ladeoperation als Sprung in den Ausnahmekontrollfluss führt jedoch schon zu merklichen Verbesserungen.

5.2.2 Programmperformanz

Gerade die Überprüfung der Reihungsgrenzen scheint teuer, wie auch die wiederholte Messung (mit Laufzeitunterstützung) von „Queens“ bei partiell abgeschalteter Implementierung (siehe Tabelle 5.3) zeigt. Die zur Überprüfung des Nullzeigers eingeführten Ladeoperationen scheinen dagegen nicht so stark ins Gewicht zu fallen. Interessant ist auch die Steigerung, die durch gleichzeitiges Deaktivieren beider Ausnahmesituationen entsteht. Da so große Stücke des

ohne *NullPointerException*	47.4s
ohne *ArrayOutofBoundsException*	40.8s
ohne *NullPointerException* und *ArrayOutofBoundsException*	35.9s

Tabelle 5.3: Selektives Abschalten von Ausnahmeunterstützung bei Queens

Programms ohne mögliche Ausnahmen optimiert werden können, scheint hier besserer Kode erzeugt zu werden. Merkwürdigerweise ist die gemessene Laufzeit sogar geringfügig geringer, als die des Kompilats mit komplett abgeschalteter Ausnahmebehandlung, Die Implementierung der Ausnahmebehandlung scheint jedenfalls im Allgemeinen keinen negativen Effekt zu besitzen.

Das Programm „Exceptions" löst 10 Millionen Ausnahmen [1] aus. Hier zeigt sich, dass die Ausnahmebehandlung über Signale trotz des stark erhöhten Aufwands noch verhältnismäßig schnell ist. Der zweite Summand in der Tabelle bezeichnet dabei den Anteil an der Laufzeit, der zusätzlich durch für die vom Betriebssytem durchgeführte Signalbehandlung entsteht. Der übrige Zusatzaufwand entsteht durch die Suche in der Ausnahmetabelle. Es konnten durchschnittlich trotzdem noch ca. 500 000 Ausnahmen pro Sekunde behandelt werden. Die frühe explizite Modellierung des Ausnahmekontrollflusses scheint hier bemerkenswerterweise sogar von Nachteil.

5.2.3 Ergebnisse

Die abstrakte Darstellung macht sich in Bezug auf die Übersetzerlaufzeit bezahlt. Gleichzeitig sind keine Optimierungsvorteile durch die alternative explizite Darstellung erkennbar.
Momentan nimmt die Laufzeit sowohl des Übersetzers als auch des übersetzten Programms durch die Ausnahmebehandlung stark zu. Optimierungen, welche unnötige Ausnahmen entfernen, könnten hier in beiden Fällen Abhilfe schaffen. Gerade die Optimierung von Reihungen innerhalb von Zählschleifen (zB. nach [MGM99]) scheint hier ein wichtiger Ansatzpunkt.

[1] **ArithmeticException**,**NullPointerException**,**ArrayOutOfBoundsException** und eine benutzerdefinierte Ausnahme, welche einen Aufruf terminiert, jeweils 2,5 Mio mal

Kapitel 6

Zusammenfassung

Es wurden die Rahmenbedingungen für die Implementierungen der Ausnahmebehandlung in einem optimierenden Javaübersetzer erörtert. Beim Aufbau wurde eine knappe Darstellung entwickelt, welche größtmögliche Freiheitsgrade für das weitere Vorgehen ermöglicht. Die Darstellung in der Zwischensprache *Firm* erlaubt dabei die Abbildung auf wenige Grundprobleme. Für javaspezifische Probleme wurden konkrete Lösungsansätze vorgestellt.

Für die Abbildung auf ein Zielsystem wurde ein zweiteiliges Vorgehen gewählt. Mit dem Wissen, dass manche Konstrukte auf dem Zielsystem keine direkten Entsprechungen haben, werden in einer ersten Phase diese an das Zielsystem angenähert. So kann erneutes Optimierungspotential geschaffen werden. Die verbleibenden Konstrukte können dann in einer zweiten Phase über die Kodeselektion effizient auf das Zielsystem abgebildet werden. Dabei werden den ohnehin im Betriebssystem und auf dem Prozessor vorhandenen Mechanismen Funktionen und Datenstrukturen zur Seite gestellt. Diese übernehmen die Rolle eines Laufzeitsystems.

Es konnt gezeigt werden, dass das entwickelte Konzept in vielerlei Hinsicht die naive Lösung durch explizite Vergleiche übertrifft. Zum einen konnte durch eine knappe Darstellung die Übersetzerlaufzeit stark reduziert werden. Zum anderen wurde das Ziel den Einfluss der Ausnhamebehandlung auf die Programmausführung zu reduzieren erreicht. Durch Ausnutzen von Mechanismen der Zielarchitektur kann das Überprüfen auf den Ausnahmezustand stark beschleunigt werden. Der reine Ausnahmekontrollfluss wird zwar dadurch verlangsamt, dies macht sich jedoch erst bei einer unrealistisch hohen Ausnhamehäufigkeit bemerkbar.

Resultat ist eine vergleichsweise schnelle, funktionierende Implementierung, welche in eine bestehende Übersetzerumgebung integriert wurde und erweiter- und portierbar ist. Zum einen wird dadurch im Jack-Übersetzer bisher fehlende Funktionalität vervollständigt. Vor allem aber stellt sie einen Ausgangspunkt und Testumgebung für weitergehende Arbeiten in Bezug auf Ausnahmenoptimierung dar.

Ob auf Grundlage dieser Implementierung ein Maximum an Performanz zu erreichen ist, muss die Implementierung von Optimierungen und Kodeerzeugern für spezielle Umgebungen zeigen. Da jedoch nur an einigen Stellen javaspezifische Konstrukte benutzt wurden und ansonsten Darstellungen gewählt wurden, auf denen bereits erfolgreich Optimierungen implementiert worden sind, sollte auch dies gelingen.

Literaturverzeichnis

[Bar00] BAR, MOSHE: *Kernel Korner*. Linux J., 2000(73es):23, 2000.

[HBS+99] HENTY, D. S., J. M. BULL, L. A. SMITH, M. D. WESTHEAD und R. A. DAVEY: *A Benchmark Suite for High Performance Java*. Oktober 1999.

[IBM04] IBM: *The Jikes Homepage*, 2004. http://www.ibm.com/developerworks/oss/jikes/.

[Int94] INTEL: *Intel386 SX MICROPROCESSOR*, 1994. Datasheet.

[Int04] INTEL: *IA32 Intel Architecture Software Developer's Manual*, 2004. Instruction Set Reference.

[MGM99] MOREIRA, JOSE E., MANISH GUPTA und SAMUEL P. MIDKIFF: *From Flop to MegaFlops: Java for Technical Computing*. Februar 1999.

[RSK+00] RYDER, BARBARA G., DONALD SMITH, ULRICH KREMER, MICHAEL GORDON und NIRAV SHAH: *A Static Study of Java Exceptions Using JESP*. Nummer 1781 in *Lecture Notes in Computer Science*, Seiten 67–81. Springer-Verlag, 2000.

[*JLS*] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java(TM) Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., Juni 2000.

[*JVMS*] LINDHOLM, TIM und FRANK YELLIN: *The Java(TM) Virtual Machine Specification (2nd Edition)*. The Java Series. Addison-Wesley, Boston, Mass., April 1999.

[TLB99] TRAPP, MARTIN, GÖTZ LINDENMAIER und BORIS BOESLER: *Documentation of the Intermediate Representation FIRM*. Interner Bericht 1999-14, Dept. of Computer Science, University of Karlsruhe (TH), Dezember 1999.

[Tra99] TRAPP, MARTIN: *Optimierung Objektorientierter Programme*. Doktorarbeit, Dept. of Computer Science, University of Karlsruhe (TH), Dezember 1999.