# Dynamische Optimierung in CPS-orientierten Zwischensprachen

Diplomarbeit

November 1994

Plamen Kiradjiev

Schneisenstr. 2
22145 Hamburg
Tel.: 040 / 678 82 94

**Betreuer:**
Prof. Dr. Joachim W. Schmidt
Dr. Martin Lehmann

Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

# Inhaltsverzeichnis


**1 Einleitung** **1**

1.1 Zielsetzung . . . . . 3

1.2 Überblick . . . . . 4

**2 Das Tycoon-Szenario** **7**

2.1 Die Tycoon-Architektur . . . . . 8

2.2 Die Programmiersprache TL . . . . . 10

2.3 Modulare Programmierung in Tycoon . . . . . 12

2.4 Persistenzabstraktion in Tycoon . . . . . 14

**3 Modulübergreifende dynamische Optimierung** **17**

3.1 Existierende Ansätze zur dynamischen Optimierung . . . . . 19

3.2 Der Tycoon-Ansatz zur globalen Optimierung . . . . . 20

3.3 Funktionsrepräsentation im Tycoon-Objektspeicher . . . . . 24

**4 Die CPS-basierte Sprache TML** **27**

4.1 Charakterisierung und Überblick . . . . . 27

4.2 Syntax- und Semantikbeschreibung von TML . . . . . 30

4.2.1 Abstrakte Syntax von TML . . . . . 30

4.2.2 Elementare Operationen . . . . . 32

4.2.3 Ausnahmen . . . . . 34

4.2.4 Das Back-End des TL-Übersetzers . . . . . 35

4.3 Abbildung höhersprachlicher Konstrukte auf TML . . . . . 35

4.3.1 Konstanten, Ausdrücke, Bindungen . . . . . 36

4.3.2 Funktionen . . . . . 37

4.3.3 Wertkonstuktoren . . . . . 38
4.3.4 Kontrollstrukturen . . . . . 38
4.3.5 Ausnahmebehandlung . . . . . 40
4.3.6 Veränderliche Bindungen . . . . . 42
4.3.7 Bindung von Funktionen an elementare Operationen . . . . . 43

**5 Optimierungstransformationen: Regeln und Realisierung 45**
5.1 Grundregeln des TL-Optimierers . . . . . 48
5.2 Bezug zu Standardoptimierungstechniken . . . . . 53
5.2.1 Optimierung von Funktionsaufrufen . . . . . 54
5.2.2 Auswertung konstanter Ausdrücke . . . . . 57
5.2.3 Konstanten-/Kopien-Propagierung . . . . . 61
5.2.4 $\eta$-Reduktion . . . . . 62
5.2.5 Vereinfachung rekursiver Funktionsdefinitionen . . . . . 62
5.3 Realisierung der Optimierungstransformationen . . . . . 63
5.3.1 Die Reduktionsphase . . . . . 65
5.3.2 Die Expansionsphase . . . . . 69
5.4 Laufzeit- vs. Optimierereffizienz . . . . . 78
5.4.1 Optimierungsbeispiele . . . . . 79
5.4.2 Maßnahmen zur Verbesserung der Effizienz des TL-Optimierers . . . . 85

**6 Datenbankanfragen im Tycoon-System 89**
6.1 Das allgemeine Modell eines Anfrageevaluators und der optimierende TL-Übersetzer . . . . . 89
6.2 Die Modulschnittstelle Iter: ein Mechanismus zur Iterationsabstraktion . . . . 93
6.3 Die Regeln der algebraischen Anfrageoptimierung . . . . . 96

**7 Messungen und Beurteilung 107**
7.1 Messungen an Standardtestprogrammen . . . . . 108
7.2 Messungen an großen TL-Programmen . . . . . 112
7.3 Optimierungsparameter . . . . . 115

**8 Zusammenfassung und Ausblick 119**

**A TML Datenstruktur 121**

**B Spezifikation der Optimierungsalgorithmen** **123**

B.1 Notation . . . . . 123

B.1.1 Grundlagen . . . . . 123

B.1.2 Regeln . . . . . 124

B.1.3 Abkürzungen . . . . . 124

B.2 Der Reduktionsalgorithmus . . . . . 125

B.3 Der Expansionsalgorithmus . . . . . 131

**C Transformationen der relationalen Algebra** **137**

C.1 Quantorausdrücke . . . . . 137

C.2 Konstruktive Operationen . . . . . 138

C.3 Selektive Operationen . . . . . 139

C.4 Leere Relationen . . . . . 140

C.5 Konstantes Prädikat . . . . . 140

**D Dynamische Optimierung am Beispiel von „Türme von Hanoi“** **141**

D.1 TL-Code . . . . . 142

D.2 TML-Code (nicht optimiert) . . . . . 143

D.3 TML-Code nach der dynamischen Optimierung . . . . . 150

**Literaturverzeichnis** **155**

# 1 Einleitung

Die Programmtransformation ist ein weit verbreiteter Ansatz bei der Generierung von ausführbarem Code. Sie findet sowohl während der Übersetzung von Programmiersprachen [Peyton-Jones 87], [Kelsey 89] als auch während der Anfrageevaluation insbesondere in relationalen Datenbanksystemen [Kim et al. 84], [Freytag 87], [Lockeman, Schmidt 87], [Fegaras, Stemple 91] statt. Dabei durchläuft der Quellcode unterschiedliche Zwischenrepräsentationen, die für die jeweiligen Codeanalysen und -manipulationen während der Übersetzungs- bzw. Evaluationsprozesse geeignet sind. Generelle und mächtige Codezwischenrepräsentationen erlauben durch klare Schnittstellenbildung nach oben (zur Quellsprache) und nach unten (zur Maschine) ein hohes Maß an Unabhängigkeit, Unifizierbarkeit und Wiederverwendbarkeit von großen Teilen der Übersetzer und Evaluatoren. Zusätzlich können sie durch ihre einfache Syntax und kleine Anzahl von Sprachkonstrukten zur Komplexitätsreduktion und zur Generalisierbarkeit der darin abgewickelten Codetransformationen beitragen.

Vor dem Hintergrund dieser Anforderungen ist insbesondere die Klasse der CPS-orientierten (*Continuation Passing Style*) Zwischensprachen entwickelt worden. Auf dem Lambda-Kalkül basierend weisen sie seine allgemein anerkannte Möglichkeit zur adäquaten und einfachen Abbildung verschiedener höhersprachlicher Konstrukte auf. Darüber hinaus eignen sie sich besonders gut für Codetransformationen in der Optimierungs- und Codegenerierungsphase der Übersetzer, da sie durch die sogenannten Fortsetzungen (*continuations*) den Kontrollfluß explizit kennzeichnen. Die Fortsetzungen erübrigen einerseits die für eine Optimierung notwendige Kontrollflußanalyse, andererseits tragen sie durch eine konzeptionelle Annäherung an lineare Zielmaschinensprachen (*goto*- statt *return*-Semantik) zur Vereinfachung der Codegenerierung bei. Diese Eigenschaften der CPS-orientierten Zwischensprachen erklären ihre breite Verwendung in optimierenden Übersetzern wie *Rabbit* [Steele 78], *Orbit* [Kranz et al. 86], *HARE* [Teodosiu 91] und *Standard ML of New Jersey* [Appel 89], [Appel 92].

Im Bereich der universellen Programmiersprachen (*general purpose languages*) spielt eine Homogenisierung und Vereinfachung leistungsvoller Sprachmechanismen für die Zwecke der Optimierung eine besonders wichtige Rolle. Das breite Band des Einsatzes solcher Programmiersprachen – von der Programmierung spezieller Algorithmen über die Unterstützung ganzer Klassen von Anwendungen (z.B. Datenbankanwendungen) bis hin zur Realisierung

komplexer Softwaresysteme – stellt hohe Ansprüche an die Optimierung. Moderne Programmiersprachen bieten eine Reihe von mächtigen Konzepten wie Rekursion, Polymorphie, Funktionen höherer Ordnung, strikte Typisierung [Cardelli 90], [Matthes 93] bzw. Objekt-Orientierung [Gawecki 91], [Chambers 92] an. Solche Konzepte sollten nicht durch eine geringe Performanz erkauft werden. Deshalb empfiehlt sich hier eine geschickte Mischung von anspruchsvoller interner Programmrepräsentation und einer geeigneten Auswahl effizienter Optimierungsmethoden.

Eine wichtige Eigenschaft universeller Programmiersprachen ist das Bestreben nach System- und Maschinenunabhängigkeit. Dabei können die Optimierungen keine maschinenspezifischen Informationen ausnutzen und von der konkreten Maschinenarchitektur abstrahieren. Dieses ist ein anderer Grund für die Wahl einer Zwischensprache als Grundlage zur Durchführung der Optimierungstransformationen vor der konkreten Codegenerierung.

Persistente Sprachen und Systeme heben die Bedeutung des Optimierers entscheidend hervor. Im Gegensatz zu Datenbankprogrammiersprachen wie Pascal/R [Schmidt 77] und DBPL [Schmidt, Matthes 92], wo die Persistenz optional und explizit deklariert werden kann, kennen Sprachen wie Napier88 [Dearle et al. 89], P-Quest [Müller 91] und TL [Matthes 92a], [Matthes 92b], [Matthes 93] keine ausschließlich temporären Objekte – alle Operationen greifen auf den Objektspeicher zu. Daraus resultiert ein entscheidender Effizienzeinfluß eines jeden Speicherzugriffs und die Bemühung, überflüssige Datenzugriffe aller Art bei der Optimierung zu eliminieren. Auf der anderen Seite stellen die persistenten reflektiven Programmiersprachen eine Umgebung zur dauerhaften Haltung von Systeminformationen zur Verfügung, die den Übersetzer und speziell den Optimierer unterstützen können.

In diesem Zusammenhang ist die Frage der Anwendbarkeit der aus dem Übersetzerbau bekannten Optimierungstechniken auf die Optimierung von Datenbankanfragen interessant. Selbstverständlich kann hier nicht erwartet werden, daß Beta-Reduktion, Funktionsexpansion, Datenflußanalyse und globale Optimierungstransformationen die Rolle der Anfrageoptimierung auf der logischen Ebene unter Einbeziehung des Wissens über das Datenbankschema vollständig übernehmen können. Vielmehr sollte sich die Optimierung auf die konkrete prozedurale Datenbankanfrageimplementation und deren Effizienz in persistenten Sprachen konzentrieren.

Zum guten Programmierstil gehört die Modularisierung mit ihrem anerkannten Beitrag zur besseren Lesbarkeit, zur Wartung und zur sauberen Struktur von Programmen. Diese Methodologie weitet sich auch auf den gesamten Systementwurf als sogenannter *add-on*-Ansatz [Matthes, Schmidt 91] aus. Im Gegensatz zum *built-in*-Ansatz, wo ein großer Teil der Systemfunktionalität fest verdrahtet ist, besteht ein *add-on*-System aus einer Reihe von Bibliotheken. Dadurch werden Voraussetzungen für die Austauschbarkeit von Komponenten, Systemerweiterungen und für die Systemwartung geschaffen. Auf der anderen Seite entsteht gerade durch die Modularität ein erhebliches Problem für den *statischen* Optimierer, der während der separaten Übersetzung der einzelnen Moduln nichts über modulfremde Funktionen weiß. Je modularer das Programm ist, desto weniger Informationen stehen dem

Optimierer zur Verfügung. Es ergibt sich der Bedarf nach einer globalen Sicht über alle von einem Programm benutzten Ressourcen, die zum Übersetzungszeitpunkt nicht ohne weiteres gegeben ist.

In der Datenbankwelt wird diese globale Sicht durch die dynamische Bindung (*late binding*) an die Datenbankobjekte erreicht. Aus diesem Grund besitzen Datenbanksprachen einen interpretativen Charakter bei der Anfrageevaluation, und der Schwerpunkt der Optimierung verlagert sich von der statischen auf die dynamische Seite, d.h. hin zu dem Zeitpunkt, wo Informationen über die aktuelle Bindung und den aktuellen Datenbestand vorhanden sind: zur Laufzeit ([Ullman 88a], [Ullman 88b]).

Analog sind bei modularen Programmiersprachen die optimierungsrelevanten Informationen erst nach dem Binden über Modulgrenzen hinweg vorhanden, so daß die Optimierung in diesem Fall erst zu diesem Zeitpunkt wirklich sinnvoll ist. Um dieses zu verwirklichen, braucht das System eine Reflektionsmöglichkeit: der schon erzeugte Objektcode muß zurück in die Zwischensprache umgewandelt, dort optimiert und neu generiert werden [Srivastava, Wall 92]. Die *dynamische* Optimierung, d.h. die Optimierung nach dem Binden, verspricht, durch Anwendung verschiedener codeverbessernder Transformationen einen bedeutenden Performanzgewinn bei modernen modularen Programmiersprachen zu erzielen (s. die Messungen in Kapitel 7).

## 1.1 Zielsetzung

In der vorliegenden Arbeit werden Probleme, Methoden und Implementationstechniken der dynamischen Optimierung in der persistenten Programmierumgebung *Tycoon* [Matthes 92a], [Matthes 93] diskutiert. Konkret sollen folgende Ziele verfolgt werden:

1. Es soll ein Optimierungsmodell entwickelt und implementiert werden, das Optimierungstechniken wie *Reduktion* und *Funktionsexpansion* (*Inlining*) für die persistente Programmiersprache TL (*Tycoon Language*) in einem dynamischen Kontext auf der Ebene einer CPS-orientierten Zwischenrepräsentation anwendet. Dabei wird besonders auf Effizienz, Generalität und Wiederverwendbarkeit des Optimierers Wert gelegt.

2. Neben der Beschleunigung der Ausführung traditioneller Anwendungsprogramme sollen sich die Optimierungen auch auf die Möglichkeit zur Benutzung von TL als Datenbankprogrammiersprache richten. Die Aufmerksamkeit soll sich vor allem auf Werkzeuge und Mechanismen zur Iterationsabstraktion in Tycoon konzentrieren und auf deren Zusammenspiel mit Programmoptimierungstechniken, die im Kontext einer persistenten Programmierumgebung angewendet werden.

3. Auf der Grundlage verschiedener Tests sind die Auswirkungen der Optimierungen auf die Performanz, Codegröße und Übersetzungszeit zu analysieren. Dabei wird angestrebt, eine Parallele zwischen den Aufgaben der Übersetzungsprozesse in TL und

denen eines allgemeinen Evaluators für Datenbankanfragen zu ziehen, um die gegenseitigen Auswirkungen in persistenten Systemen zu untersuchen und die Vorteile der allgemeinen Funktionsoptimierung auch im Bereich datenintensiver Anwendungen zu nutzen.

4. Durch die ausführliche Analyse der dynamischen Optimierungskomponente von Tycoon sollen Grundlagen zum weiteren Ausbau der Optimierungsstrategien geschaffen werden. Ein Schwerpunkt liegt dabei auf der Verlagerung spezieller logischer Anfrageoptimierungen auf die allgemeine Programmoptimierung.

## 1.2 Überblick

Zunächst wird in Kapitel 2 die allgemeine Umgebung für die Optimierungsanalysen und -transformationen vorgestellt: das persistente Programmiersystem Tycoon (***Typed Communicating Objects in Open Environments***) [Matthes 92a], [Matthes 93]. Die Schwerpunkte liegen dabei auf der Systemarchitektur (Abschnitt 2.1) und auf den innovativen Konzepten der Sprache TL (*Tycoon Language*) [Matthes 92b], [Matthes et al. 94] (Abschnitt 2.2). In Abschnitt 2.3 wird das Modularitätsprinzip in Tycoon erläutert. Anschließend werden optimierungsrelevante Aspekte der Objektspeicherschnittstelle präsentiert (Abschnitt 2.4).

Die Konzepte der dynamischen Optimierung werden in Kapitel 3 diskutiert. Zunächst wird eine Übersicht gegeben über existierende Methoden zur Optimierung nach dem Binden (Abschnitt 3.1), danach folgt die Beschreibung der Reflektion im Tycoon-System sowie der Vergleich zu den bestehenden Methoden (Abschnitt 3.2). In Abschnitt 3.3 wird näher auf die Laufzeitrepräsentation von Funktionen in Tycoon eingegangen.

Kapitel 4 stellt eine Beschreibung der in dieser Arbeit verwendeten CPS-orientierten Zwischenrepräsentationssprache TML (*Tycoon Machine Language*) [Gawecki, Matthes 94] dar. Nach einem Überblick in Abschnitt 4.1 werden die Syntax und Semantik (Abschnitt 4.2) sowie die Abbildung der höhersprachlichen TL-Konstrukte in Abschnitt 4.3 präsentiert.

Das eigentliche Optimierungskonzept sowie seine Realisierung werden im Kapitel 5 behandelt. Zunächst werden die theoretischen Grundlagen des Optimierungsmodells durch die Grundregeln (Abschnitt 5.1) und die dadurch ermöglichten Optimierungstechniken (Abschnitt 5.2) erläutert. Abschnitt 5.3 stellt eine Beschreibung der technischen Realisierung der Optimierungsregeln für Reduktion und Funktionsexpansion dar, während Abschnitt 5.4 auf einige Gesichtspunkte der Effizienz des generierten Codes und der Effizienz des Optimierers selbst eingeht.

Kapitel 6 konzentriert sich auf die Auswirkungen der traditionellen Strategien für die Programmoptimierung auf die Datenbankprogrammierung und die Anfrageevaluation. In Abschnitt 6.1 wird ein allgemeines Modell eines Anfrageevaluators vorgestellt. Eine Analogie zwischen der Architektur des optimierenden TL-Übersetzers und diesem Modell leitet sich aus vielen Ähnlichkeiten der Übersetzungs- und Anfrageauswertungsvorgänge ab. Auf

Grundlage des in der Tycoon-Standardbibliothek bereitgestellten Moduls *Iter* zum Zweck der Iterationsabstraktion (Abschnitt 6.2) wird eine ausführliche Sammlung von Regeln zur algebraischen Anfrageoptimierung zusammengestellt, die auf Optimierbarkeit hin überprüft werden (Abschnitt 6.3). Davon ausgehend werden Schlußfolgerungen über Möglichkeiten und Eignung des Tycoon-Systems und des TL-Optimierers für die Anfrageoptimierung auf der physikalischen Ebene gezogen.

In Kapitel 7 wird der dynamische TL-Optimierer verschiedenen Tests unterworfen, die sowohl typische aus der Literatur bekannte Beispiele umfassen als auch zwei größere reale Programme, nämlich den TL-Übersetzer und den TL-Parsergenerator. Dabei wird die Performanzsteigerung (Abschnitt 7.1), neben Codegröße und Übersetzungsdauer (Abschnitt 7.2) im absoluten und relativen Vergleich zu den nicht optimierten bzw. statisch optimierten Versionen der jeweiligen Testprogramme analysiert. Die Steuerung des Optimierers über einen Satz von Parametern wird in Abschnitt 7.3 diskutiert.

Das letzte Kapitel bietet schließlich eine Zusammenfassung der Ergebnisse und einen Ausblick auf weitere Optimierungsverfahren, die über die Fähigkeiten des aktuell realisierten Optimierers hinausgehen.

# 2 Das Tycoon-Szenario

Dieses Kapitel stellt die persistente Programmierumgebung Tycoon (***Ty**ped **C**ommunicating **O**bjects in **O**pen **En**vironments*) [Matthes 92a], [Matthes 93] vor, welche die allgemeine Entwicklungsumgebung für den in der Arbeit beschriebenen Optimierer bildet.

Tycoon beruht auf Typsystemen höherer Ordnung und ist genügend ausdrucksvoll, um in einem unifizierten sprachlichen Rahmen verschiedene generische Dienste integrieren, erweitern, spezifizieren, verwenden und neu definieren zu können. Es vereinigt die Vorteile der Datenbankprogrammiersprachen wie Pascal/R [Schmidt 77] und DBPL [Schmidt, Matthes 92] sowie der persistenten Systeme wie Napier88 [Dearle et al. 89] und P-Quest [Müller 91]. Von den ersteren erbt Tycoon die orthogonale Kombinierbarkeit elementarer Basiskonzepte für die Persistenzabstraktion, die typvollständige Datenrepräsentation und die Iterationsabstraktion. Die Idee zur Nutzung multipler Programmrepräsentationen zu dynamischen Optimierungszwecken ist ebenfalls der Technologie integrierter Datenbankprogrammiersysteme entlehnt.

Von den persistenten Sprachen und Systemen stammt das Konzept der Abwicklung aller Speicherzugriffe (auf Daten und Programme) über eine wohldefinierte Schnittstelle auf einem sehr niedrigen Abstraktionsniveau. Dieses Konzept trägt zu einer bedeutenden Vereinfachung der Gesamtkomplexität des Systems sowie zu seiner Offenheit gegenüber verschiedenen Objektspeichern bei. Dadurch bietet Tycoon sowohl eine hohe Universalität bei der Entwicklung persistenter Anwendungen als auch ein hohes Maß an Erweiterbarkeit und Austauschbarkeit von Systemkomponenten.

Einen besonderen Vorteil weist Tycoon bei der Integration und Verwendung verschiedener Diensterbringer auf, die persistente Systeme in ihrer Langlebigkeit, Benutzernähe und ihren Sicherheitsanforderungen unterstützen. Abb. 2.1 stellt die Kopplung von drei Diensterbringern dar: einem Datenbanksystem (z.B. Ingres), einer Programmiersprache (z.B. C) und einem Bildschirmserver (z.B. Sun-NeWS). Während die übliche Kommunikation zwischen den Diensterbringern über sehr schmale Schnittstellen (Zeichenketten, Cursor, Zeiger) erfolgt, bietet Tycoon einen gemeinsamen Benennungs-, Bindungs- und Typisierungsmechanismus, so daß die Objekte verschiedenartiger externer Dienste in einem einheitlichen Kontext verwendet werden können. Diese gemeinsame Sicht trägt entscheidend zur Flexibilität, Effizienz und Korrektheit ihres Zusammenspiels bei.

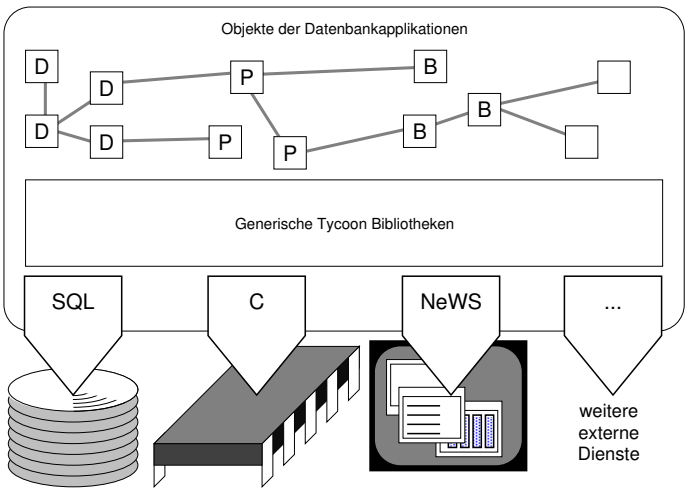



Abbildung 2.1: Kopplung von Diensterbringern in Tycoon (aus [Matthes 92a])

Ein wichtiges Merkmal des Tycoon-Systems im Vergleich zu anderen Ansätzen für Datenbankprogrammierung ist, daß es keine Aufwärtskompatibilität mit existierenden Programmiersprachen oder Datenmodellen beibehält. Seine internen Protokolle für Speicherzugriffe, Programmrepräsentation und Binden haften an keinerlei vorexistierenden Standards. Vielmehr bietet es Abstraktionen zur Definition höherer, problemorientierter Modelle an und agiert somit als ein *lean*-Service [Schmidt, Matthes 93] zur Anbindung generischer Dienste.

Im folgenden wird ein Überblick über das Tycoon-System angeboten. Als erstes (Abschnitt 2.1) wird die Architektur des Systems präsentiert. In Abschnitt 2.2 folgen die wichtigsten Konzepte der universellen Programmiersprache TL. Abschnitt 2.3 geht näher auf die modulare Programmierung im Tycoon-System ein. Anschließend wird kurz das Konzept der Objektspeicherschnittstelle vorgestellt (Abschnitt 2.4).

## 2.1 Die Tycoon-Architektur

Das Tycoon-System ist als eine Schichtenstruktur gestaltet (Abb. 2.2). Es gibt folgende Schnittstellen zwischen den einzelnen Schichten:

**TL-Schnittstelle:** Die Programmiersprache TL (*Tycoon Language*) ist eine algorithmisch vollständige, strikt typisierte, modulare universelle Programmiersprache, die Funktionen und Moduln als Objekte erster Ordnung behandelt. TL kann nicht nur zur Datenmodellierung und Anwendungsentwicklung eingesetzt werden, sondern liefert einen sprachenunabhängigen Rahmen für Programmübersetzung, -generierung und -bindung, wodurch sie sich selbst als Systemsprache über erweiterbare Bibliotheken

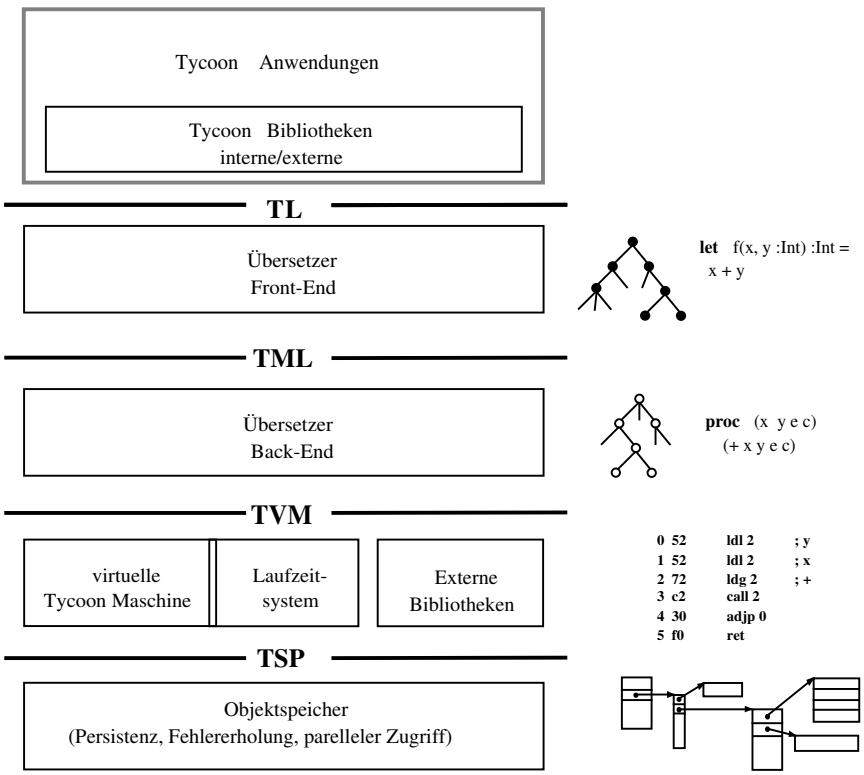



Abbildung 2.2: Tycoon-Schichtenarchitektur

und Sprachprozessoren implementiert. Durch die Uniformität der Programmierumgebung, die potentielle Persistenz aller Objekte und die strikte Typisierung bietet die Sprache TL qualitativ neue Dimensionen zur generischen Programmierung in verschiedenen Stilen, z.B. funktional, imperativ, objekt-orientiert oder logik-basiert [Matthes 92a], obwohl sie auf keines der Modelle speziell ausgerichtet ist. Insbesondere eignet sie sich für die Benutzung als Datenbanksprache. Technisch gesehen ist TL ein typisierter Lambda-Kalkül zweiter Ordnung, der um Zuweisungen, dynamische Typen und Subtypisierungsregeln erweitert ist [Cardelli 90], [Matthes 92b]. Dies deutet bereits auf ihre Eignung zur Transformation in eine Lambda-Kalkül-basierte Zwischenrepräsentation hin [Peyton-Jones 87].

**TML-Schnittstelle:** Die interne Sprache TML (*Tycoon Machine Language*) ist ein CPS-basierter untypisierter Lambda-Kalkül. Die wichtigsten Vorteile von TML sind die deutliche Reduktion der vom Optimierer bzw. Codegenerator zu handhabenden Programmkonstrukte und die explizite Angabe des Kontrollflusses, die ihrerseits die Datenflußanalyse vereinfacht. Eine genaue Beschreibung von TML folgt in Kapitel 4.

**TVM-Schnittstelle:** TVM (*Tycoon Virtual Machine*) ist eine lineare Bytecoderepräsentation von Instruktionen, die für eine effiziente Interpretation von einer abstrakten Maschine geeignet ist. Sie stellt das ausführbare Programm dar, das nach der Optimierungsphase vom *Back-End* des Übersetzers erzeugt wird.

**TSP-Schnittstelle:** Auf der untersten Ebene befindet sich das datenmodellunabhängige Software-Protokoll TSP (*Tycoon Store Protocol*), das verschiedene Objektspeicherimplementationen und ihre Komponenten einkapselt. Dadurch bietet es den höher liegenden Systemschichten eine Abstraktion von den Prozessen der Zugriffsoptimierung, Speicherallokation und -deallokation, Nebenläufigkeit und Verteilung der Daten innerhalb des physikalischen Speichers. In Abschnitt 2.4 wird näher auf einige optimierungsrelevante Aspekte dieser Schnittstelle eingegangen.

Wie in Abb. 2.2 angedeutet, interagieren die Tycoon-Komponenten folgendermaßen: Die Anwendungen werden in der höheren Programmiersprache TL durch die Unterstützung einer Reihe von Bibliotheken geschrieben. Das *Front-End* des Übersetzers übernimmt die Aufgabe des Parsens und der Erzeugung eines abstrakten Syntaxbaumes, auf dem wiederum die statische Typüberprüfung vorgenommen wird. Anschließend wird der typkorrekte Programmcode in eine TML-Baumstruktur umgewandelt, worin sich die ganzen Optimierungsanalysen und -transformationen abspielen. Der TML-Code wird schließlich vom *Back-End* des Übersetzers in den interpretierbaren TVM-Bytecode transformiert. Unter Verwendung der TSP-Schnittstelle wird dieser in den Objektspeicher abgelegt und vom Laufzeitsystem ausgeführt.

## 2.2 Die Programmiersprache TL

Die Tycoon-Systemsprache TL kann als eine Weiterentwicklung von Quest [Cardelli 90] und P-Quest [Müller 91] angesehen werden, wobei sie mit polymorphen funktionalen Sprachen der ML-Familie verwandt ist, gleichzeitig aber in ihrer Modulstruktur und den imperativen Elementen den Sprachen aus der Modula-Familie ähnelt. Dadurch stellt TL eine Symbiose aus imperativen und funktionalen Programmiersprachen dar.

Da eine ausführliche Beschreibung von TL in dieser Arbeit nicht möglich ist, wird dazu die spezielle Tycoon-Literatur empfohlen. Die Grundkonzepte des Tycoon-Systems und der Sprache TL sind in [Matthes 92a] bzw. [Matthes 93] dargelegt. In [Matthes et al. 94] ist eine detaillierte Einführung in TL zu finden.

Im folgenden werden die charakteristischen Eigenschaften von TL aufgezählt, um eine globale Übersicht über diese moderne universelle Programmiersprache zu geben:

- In TL existieren keine vordefinierten Basistypen, Konstanten bzw. Funktionen. Sie werden von Moduln der Tycoon-Standardbibliothek importiert und in der initialen Umgebung an gängige Namen und symbolische Bezeichner gebunden.

- Durch unveränderliche *Bindungen* werden Objekte mit benutzerdefinierten Namen zu ihrer weiteren Verwendung in Ausdrücken bezeichnet.
- *Zustandsvariablen* werden explizit durch veränderliche Bindungen gekennzeichnet, durch die immer ein Initialisierungswert angegeben wird.
- Rekursive Bindungen werden explizit gekennzeichnet und statisch auf Korrektheit überprüft.
- Die vordefinierten Wertkonstruktoren sind auf fünf primitive Konstruktoren reduziert, die orthogonal miteinander kombinierbar sind: Tupel, variantes Tupel, Record, Feld und Ausnahme (*exception*). Die Tycoon-Standardbibliothek stellt durch die geschachtelte und rekursive Anwendung dieser Primitive zusammen mit den Funktionen eine breite Palette an Typkonstruktoren zur Modellierung von Mengen, Listen, Relationen, Assoziationen, Klassen, Dateien, etc. sowie polymorphe Funktionen zur Manipulation von Werten dieser parametrisierbaren abstrakten Typen zur Verfügung.
- Aufgrund der wenigen vordefinierten Wertkonstruktoren reichen die üblichen *if*- und die *case*-Kontrollstrukturen zu ihrer Manipulation aus. Außerdem werden drei Schleifenkonstrukte sowie Mechanismen zur Ausnahmebehandlung angeboten.
- Funktionen stellen Objekte erster Ordnung dar. Dementsprechend können sie an andere Funktionen als Argumente übergeben bzw. als Resultate zurückgegeben werden.
- Bei Funktionsaufrufen gibt es die Möglichkeiten zur Wertparameter- und Referenzparameterübergabe.
- Durch *Signaturen* wird Typinformation statisch einem Namen zugeordnet. Parallel gibt es die Möglichkeit zur Typinferenz.
- Typen können auch an Namen gebunden werden. Sie können als Werte höherer Ordnung aufgefaßt werden.
- Neben der statischen gibt es in TL auch die dynamische Typbindung, die durch Typoperatoren realisiert ist.
- Es existieren zwei Ausprägungen des Polymorphismus in TL: parametrischer und Subtyppolymorphysmus. Polymorphe Funktionen stellen ein grundlegendes Werkzeug beim Entwurf der Tycoon-Bibliotheken dar.
- Zwischen vordefinierten Typkonstruktoren und Typoperatoren sind strukturelle Subtypbeziehungen definiert.

Vom Standpunkt der dynamischen Optimierung aus gesehen sind jedoch nicht alle TL-Eigenschaften von Bedeutung. Wegen der statischen Typüberprüfung (im *Front-End* des Übersetzers) konzentriert sich das Interesse bei den Zwischencodetransformationen vor allem

auf Wertbindungen und -konstruktoren sowie auf Kontrollstrukturen. Die Persistenz und die ausgeprägte Modularität der Sprache TL üben einen besonderen Einfluß auf den TL-Optimierer aus. Aus diesem Grund werden die Beziehungen zwischen diesen zwei zentralen Tycoon-Eigenschaften im nächsten Abschnitt näher erläutert.

## 2.3 Modulare Programmierung in Tycoon

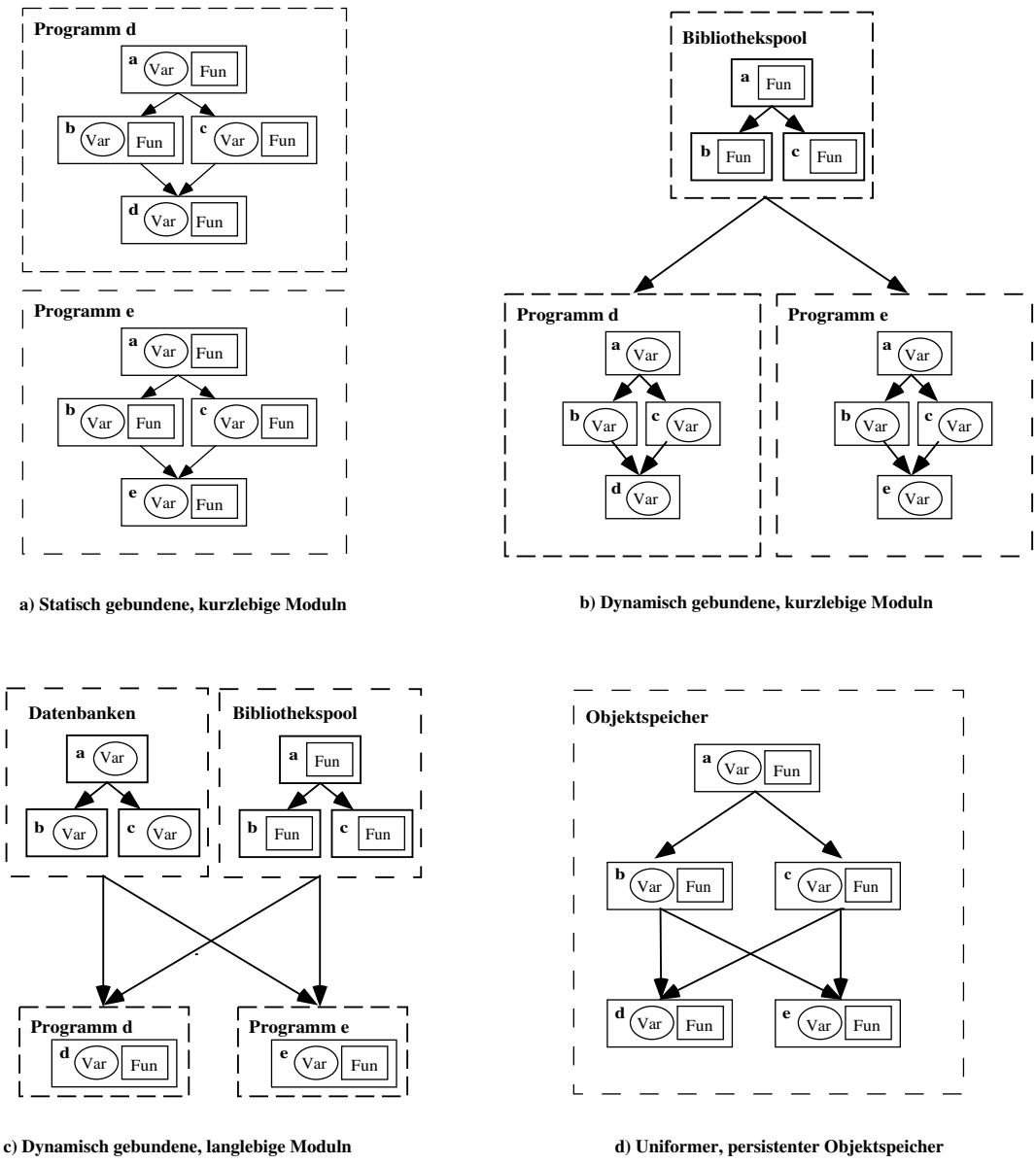
Als eine moderne Programmiersprache bietet TL das Modulkonzept zur Strukturierung des Namensraums an. Dadurch werden Konsistenzbedingungen über Komponentengrenzen hinweg erzwungen, ohne den Übersetzungsaufwand für eine Übersetzungseinheit mit der Komplexität des gesamten Systems zu belasten.

TL kennt den Modul (**module**), die Modulschnittstelle (**interface**) und die Bibliothek (**library**) als Modularisierungseinheiten. Sie stellen keine Erweiterung der Benennungs-, Bindungs- und Typisierungsschemata dar, vielmehr schränken sie die Orthogonalität existierender Konzepte (Funktionen, Tupeltypen, geschachtelte Sichtbarkeitsbereiche) ein. Diese Restriktionen vereinfachen die werkzeugunterstützte Entwicklung und Wartung großer Softwaresysteme und erlauben die Interaktion mit externen Diensten, die typischerweise einfachere Namens- und Bindungsparadigmen besitzen.

Bibliotheken legen einen globalen Sichbarkeitsbereich für Modul- und Schnittstellennamen sowie für andere Bibliotheken fest. Modulschnittstellen sind benannte Tupeltypen, die sich auf andere im globalen Sichtbarkeitsbereich bekannten Moduln und Schnittstellen beziehen können. Sie stellen wiederum anderen Moduln und Schnittstellen Signaturen zur Verfügung. Moduln ihrerseits werden durch einen benannten Tupelwert vom Typ der jeweiligen Schnittstelle repräsentiert. Damit stellt die Verwendung der darin definierten Bindungen und Funktionen einen üblichen Tupelzugriff dar. Von der für den Optimierer relevanten Implementationssicht also beruhen die Modularisierungseinheiten in TL auf dem Tupelkonzept, so daß sie einer einheitlichen Behandlung mit den „konventionellen“ Wertkonstruktoren unterzogen werden können.

Im persistenten Tycoon-System wird das Binden der Moduln durch Bekanntmachung (Import) der einzelnen modulbeschränkten Sichtbarkeitsbereiche auf dem obersten Sichtbarkeitsbereich (*Tycoon Top Level*) realisiert. Die technische Realisation sieht folgendermaßen aus: Zur Übersetzungszeit wird jeder Modul in eine Funktion transformiert, die einen Tupelwert vom Typ der jeweiligen Schnittstelle liefert. Beim Import des Moduls wird diese Funktion ausgeführt, indem ihr Ergebniswert im Objektspeicher abgelegt und an den jeweiligen Modulnamen im globalen Sichtbarkeitsbereich gebunden wird. Dieser Prozeß stellt in Tycoon das Binden (*Linking*) dar.

Die Rolle der Persistenz im Tycoon-System drückt sich in der Art und Weise der Bindung von Moduln im globalen Sichtbarkeitsbereich aus. Anders als in Systemen mit flüchtigen Bindungen, wo jedes Programm eine Kopie der importierten Bibliotheken erhält, und die




a) Statisch gebundene, kurzlebige Moduln

b) Dynamisch gebundene, kurzlebige Moduln

c) Dynamisch gebundene, langlebige Moduln

d) Uniformer, persistenter Objektspeicher

Abbildung 2.3: Bindung und Lebensdauer in modularen Systemen

Seiteneffekte auf die in diesen Bibliotheken definierten globalen Variablen lokal bleiben, wird in Tycoon eine einmalige Initialisierung der von verschiedenen Programmen verwendeten Moduln durchgeführt, wobei die Änderungen auf globale Variablen überall sichtbar werden. Abb. 2.3 stellt die Konzepte der statischen (a) und dynamischen (b) Bindung kurzlebiger Moduln der dynamischen Bindung langlebiger Moduln in Datenbankprogrammiersprachen (c) und dem uniformen Bindungskonzept in persistenten Objektsystemen (d), wie Tycoon selbst, gegenüber. Die Pfeile drücken die Importbeziehungen aus, indem sie von dem Definitions- zum importierenden Modul zeigen. In den ersten zwei Fällen (a und b) werden die global definierten Variablen mit einer Kopiersemantik den beiden Programmen *d* und *e* zur Verfügung gestellt, wobei im Fall dynamischer Bibliotheken (*dynamic shared libraries*) die Funktionen von einem gemeinsamen Pool aus benutzt werden.

Der Vorteil bei den persistenten Systemen im Vergleich zu den Datenbanksystemen ist, daß sie einen unifizierten Persistenzbegriff sowohl über Daten als auch über Moduln besitzen: Daten, Funktionen und Programme werden einheitlich als Speicherobjekte behandelt. Diese Orthogonalität von Sprachkonstrukten und Persistenz erlaubt die Manipulation von Objekten beliebiger Struktur und übertrifft die Möglichkeiten von Datenbankprogrammiersprachen wie Pascal/R und DBPL, die die Persistenz auf spezielle Konstrukte (z.B. Relationen) einschränken.

## 2.4 Persistenzabstraktion in Tycoon

Eine wichtige Rolle bei der Implementation des dynamischen TL-Optimierers spielt die durch das TSP-Protokoll ermöglichte Persistenzabstraktion. Auf der untersten Systemebene bietet dieses Protokoll die Grundfunktionalität eines unbegrenzten Objektspeichers, der das Anlegen, Lesen und die Modifikation von Speicherobjekten gestattet.

Jedes Objekt besitzt eine Identität. Die Persistenz der Objekte setzt ihre Erreichbarkeit vom Wurzelobjekt des Objektspeichers voraus. So sind die im globalen Sichtbarkeitsbereich definierten TL-Bindungen von einem direkt unterhalb des Wurzelobjekts liegenden Zustandsvektor (*Top Level Vector*) referenziert. Das TSP-Protokoll bietet die Funktionalität zum Setzen von Sicherungspunkten bzw. zur Entfernung unerreichbarer Objekte (*Garbage Collection*).

Eine wichtige Überlegung bei der Entwicklung des dynamischen Optimierers ist die Tatsache, daß Objektreferenzen innerhalb eines Objektspeichers nicht konstant bleiben müssen. Bei einer Restrukturierung des Objektspeichers können sie modifiziert werden. Auf der einen Seite obliegt die Wahrung der referenziellen Integrität der Implementation des TSP-Protokolls, auf der anderen muß diese Veränderlichkeit vom Optimierer berücksichtigt werden. Bei der Verwaltung von Speicherobjektinformationen darf er sich nicht auf die Gleichheit der Objektreferenzen verlassen, da sie selbst während der Optimierungsphase modifiziert werden könnten (z.B. bei einer automatischen *Garbage Collection*).

TSP kennt folgende zwei elementare Objektformate (TSP-Felder):

- das byteorientierte Feld
- das wortorientierte Feld

Ihre Einfachheit und Generalität erlaubt die Anbindung verschiedener Objektspeicher an das Tycoon-System auf unterschiedlichen Hardware-Architekturen. Tycoon-Anwendungen können diese zwei Objektformate benutzen, um verschiedene Werte zu repräsentieren.

Byteorientierte Felder eignen sich gut zur Repräsentation von Zeichenketten und reellen Zahlen. Die wortorientierten Felder stellen Zustandsvektoren dar, deren Elemente einen beliebigen direkten Wert enthalten können. Solche können ganze Zahlen, wortausgerichtete Hauptspeicheradressen und wiederum Objektreferenzen sein. Dieses Format bietet ein mächtiges Mittel zur Darstellung der unterschiedlichen Wertkonstruktoren, Funktionen und Moduln im Objektspeicher.

Jedem TSP-Feld wird ein Deskriptor zugewiesen, der u.a. Informationen über Objektgröße und -mutabilität enthält. Diese Informationen beziehen sich auf das ganze Feld. Für den Optimierer wäre detaillierteres Wissen über die Mutabilität der Feldelemente eher von Vorteil. Zur Zeit bietet das TSP-Protokoll den Kompromiß der groberen Mutabilitätsinformation zugunsten eines kleineren Verwaltungsaufwands an, wobei ein Feld dann als unveränderbar gekennzeichnet wird, wenn es gar keine veränderbaren Elemente enthält.

Die Abstraktion des TSP-Protokolls von operationellen Qualitäten des Objektspeichers, wie Persistenz, Zugriffsgeschwindigkeit, Nebenläufigkeit, Speicherallokation und -deallokation, Fehlererholung und Verteilung leistet einen wichtigen Beitrag zur Portabilität und Skalierbarkeit des Tycoon-Systems. Durch die Einhaltung des TSP-Protokolls wird die Persistenz und die konsistente Manipulation der langlebigen Speicherobjekte garantiert.

# 3 Modulübergreifende dynamische Optimierung

Moderne persistente Systemen weisen große Vorteile in Bezug auf die inkrementelle Entwicklung langlebiger Systeme auf und tragen zur hohen Wiederverwendbarkeit und Wartbarkeit ihrer Komponenten bei. Ihre herausragenden Eigenschaften wie Modularität, dynamische Bindung und Polymorphie schränken jedoch einen konventionellen lokal optimierenden Übersetzer bei der Anwendung der codeverbessernden Transformationen stark ein.

Modularität (die hohe Funktionsabstraktion und Datenkapselung erlaubt) schränkt die Codeanalyse bei einer getrennten Übersetzung auf einen kleinen lokalen Kontext ein. Der dadurch bedingte Mangel an Informationen über modulfremde Objekte hindert den Optimierer, effizienten Code für die einzelnen Übersetzungseinheiten zu erzeugen. Je feiner die Modularisierungsstruktur der Programmierumgebung wird, desto weniger Transformationen können vorgenommen werden, und umso ineffizienter bleibt der erzeugte Code. Im Extremfall degradiert diese Situation bis zur absoluten Unfähigkeit des Optimierers, den Quellcode zu verändern.

Das Zusammenspiel der Modularität und der dynamischen Bindung von Funktionen, Moduln und Bibliotheken in einem persistenten System erlaubt die konzeptionelle Trennung zwischen logischen und physikalischen Schemata. Im Tycoon-System wird dies durch mehrere einer Schittstelle zugeordnete Moduln ermöglicht, die verschiedene Implementationen zu der von der Schnittstelle festgelegten Signatur anbieten. Durch dynamisches Binden kann einer der Moduln zur Laufzeit ausgewählt werden. Der Optimierer kann also erst dann auf die konkrete Implementation zugreifen, um globale codeverbessernde Transformationen zwischen den konkreten Moduln vorzunehmen.

Ferner wirkt sich die Polymorphie nachteilig auf die Optimierungsmöglichkeiten aus. Im Tycoon-System erlaubt sie eine Iterationsabstraktion über Massendatentypen (s. Kapitel 6) mit einer einheitlichen Bearbeitung verschiedener Datenstrukturen. Auch hier bleiben sowohl die Objekte als auch ihre Manipulationsfunktionen bis zum Bindungszeitpunkt unbekannt.

Die Konsequenz daraus ist, daß in Systemen mit hoher Modularität, dynamischer Bindung und Polymorphie der Optimierer den statischen Rahmen des Übersetzers sprengen und

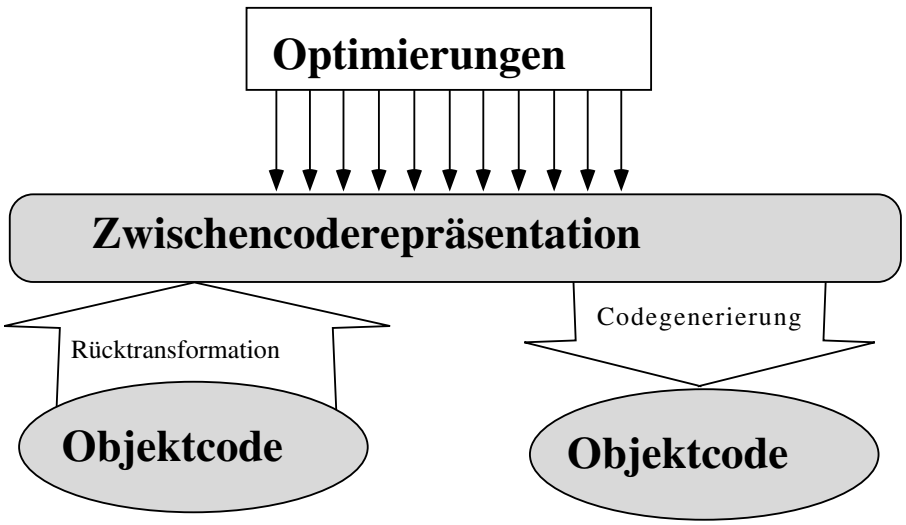



Abbildung 3.1: Das generelle Modell der dynamischen Optimierung

seine Transformationen zu einem späteren Zeitpunkt, nach dem Binden bzw. zur Laufzeit eines Programms, durchführen sollte. Damit er globale Codeanalysen und -transformationen vornehmen kann, sollte die Programmierumgebung die reflektive Möglichkeit zur Rücktransformation des ausführbaren Objektcodes der zu optimierenden Moduln besitzen. Das allgemeine Schema ist in Abb. 3.1 dargestellt.

Im Gegensatz zur *statischen Optimierung*, die zur Übersetzungszeit stattfindet, wird die *dynamische Optimierung* eines Programms nach der separaten Übersetzung der involvierten Moduln durchgeführt. Dazu sind drei Schritte erforderlich (vgl. Abb. 3.1):

1. Transformation des bei der separaten Übersetzung erzeugten Objektcodes zurück in die für die Optimierung geeignete Zwischenrepräsentation

2. Globale modulübergreifende Codeanalysen und -transformationen

3. Erneute Codegenerierung für die optimierten Programmkomponenten

In diesem Kapitel werden die Konzepte der globalen Optimierung nach dem Binden diskutiert. Zunächst werden in Abschnitt 3.1 einige existierende Ansätze präsentiert. Abschnitt 3.2 stellt eine Beschreibung des Prinzips der in Tycoon implementierten dynamischen Optimierung dar. Im letzten Abschnitt (3.3) wird die Struktur der Laufzeitrepräsentation einer Funktion im Objektspeicher von Tycoon behandelt.

## 3.1 Existierende Ansätze zur dynamischen Optimierung

In der Literatur gibt es wenige Ansätze in dieser Richtung. In [Srivastava, Wall 92] wird ein interessantes System (OM) für Bindezeit-Codeoptimierung vorgestellt, das eine zum jeweiligen Programm gehörende Kollektion von Objektcode-Moduln in die symbolische RTL-Form (*Register Transfer Language*) zurück konvertiert, dort interprozedurale Optimierungen durchführt und aus der resultierenden RTL-Repräsentation wieder Objektcode generiert. Srivastava und Wall haben auf die Vorteile der dynamischen Optimierung besonders bei den von ihnen verwendeten Optimierungstechniken – Registerallokation und Code-Verschiebung (*code motion*) – hingewiesen. Ein besonderes Merkmal des OM-Systems ist, daß es auf dem vom jeweiligen Übersetzer unabhängigen Objektcode operiert und dadurch Optimierungen nicht nur über Modulgrenzen, sondern auch über Programmiersprachen hinweg ermöglicht. Diese höhere Stufe der Abstraktion führt jedoch zu einem Informationsverlust bezüglich höhersprachlicher Konstrukte, sowie zu einer gewissen Komplexität der Rücktransformation nach RTL. Die Autoren haben ihren Ansatz auf C-Beispielen getestet und eine Performanzsteigerung von 5% gegenüber dem mit dem konventionellen MIPS-Übersetzer (mit der Option *-O2*) erzeugten Code gemessen. Der Vergleich zu den deutlich besseren Ergebnissen in Kapitel 7 bezüglich des dynamisch und statisch optimierten Codes (unter Anwendung derselben Optimierungstechniken) unterstreicht die steigende Bedeutung der dynamischen Optimierung in ausgesprochen modularen Systemen wie Tycoon.

Ein ähnlicher Ansatz wird in [Orr et al. 93] verfolgt, wo das Laden von Programmen durch einen *Meta-Objekt-Server* unterstützt wird, der die dynamische Observation von Programmen über Modulgrenzen hinweg ermöglicht. Die Autoren untersuchen dabei die sich daraus ergebenden Vorteile bei der Realisierung von Debugging- und Optimierungsstrategien. Dieser Beitrag deutet auch auf eine weitere Anwendung der Reflektion in persistenten Systemen: auf die Möglichkeit zur Implementation von dynamischen Programmonitoren und -debugger.

Besonders großen Vorteil bietet der Informationsrückfluß aus der Laufzeitumgebung bei objekt-orientierten Systemen, speziell die Gewinnung von Laufzeit-Typinformationen, die in [Hölzle, Ungar 94] als *type feedback* bezeichnet wird. Der SELF-Übersetzer, ein optimierender Übersetzer für eine rein objekt-orientierte Programmiersprache ([Chambers, Ungar 90], [Chambers, Ungar 91], [Chambers 92]), basiert auf einer Eigenschaft zur Buchführung des sogenannten Typ-Profils für jeden Methodenaufruf im Programm. Dieses wird dem Optimierer reflektiv zur Verfügung gestellt, um ihn bei der Typinferenz und der daraus resultierenden Ersetzung des Methoden-Sendens durch effizientere Prozeduraufrufe und ihre eventuelle Expansion zu unterstützen. In einer ersten Phase werden die Methoden separat unoptimiert übersetzt. Später, in Abhängigkeit von der Häufigkeit ihrer Ausführung, werden sie unter Ausnutzung der zur Laufzeit gebildeten Typ-Profile optimiert. Zum Schluß ersetzen die neuen effizienteren Methoden-Versionen die bisherigen auf dem Laufzeitstapel. Das Besondere beim SELF-Übersetzer ist, daß er automatisch über die Notwendigkeit einer Rekompilation entscheidet. Die Konzentration auf die von der Laufzeit erworbenen Typinformationen ist

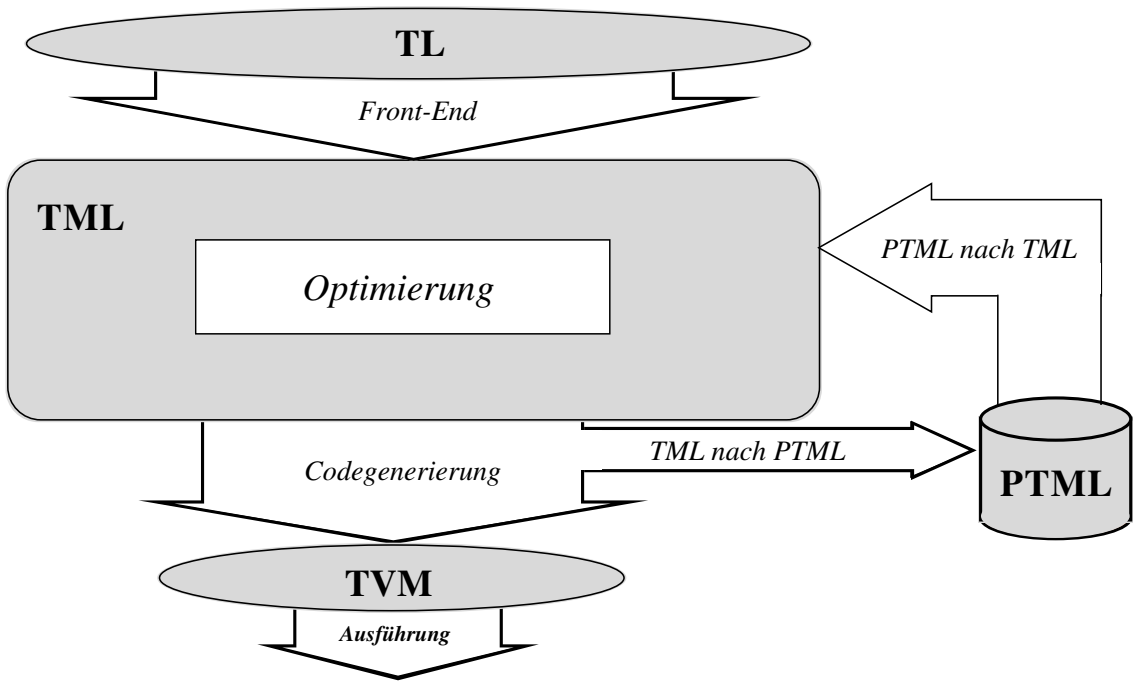



Abbildung 3.2: Reflektion in Tycoon

bei einem objekt-orientierten System wie SELF natürlich, da sie bei der Eliminierung der relativ teuren Botschaften zwischen Objekten eine zentrale Rolle spielen.

## 3.2 Der Tycoon-Ansatz zur globalen Optimierung

Bei einer modularen persistenten Programmierumgebung wie Tycoon verlangt der Optimierer eine globale dynamische Sicht auf alle von dem zu optimierenden Objekt erreichbaren Moduln und Funktionen, damit er codeverbessernde Transformationen (Funktionsexpansion, Kopien-Propagierung, Entfernen unerreichbaren Codes) über Modulgrenzen hinweg vornehmen kann. Aus diesem Grund bietet das Tycoon-System die Reflektion: die Möglichkeit zur Wiederherstellung der TML-Zwischenrepräsentation aus den ausführbaren, im Objektspeicher abgelegten Funktionsobjekten. Diese Systemeigenschaft erlaubt die Durchführung der globalen Optimierung in Tycoon nach der Bindezeit.

Der Reflektionszyklus findet während der *Back-End*-Phase (Abb. 3.2) des TL-Übersetzers statt. Die *Front-End*-Phase des TL-Übersetzers generiert typkorrekten TML-Code. Zur Übersetzungszeit kann darauf die statische Optimierung durchgeführt werden. Die Codegenerierung erzeugt aus der TML-Repräsentation den ausführbaren TVM-Bytecode, der von der Tycoon-Maschine interpretiert werden kann. Optional kann außerdem eine kom-

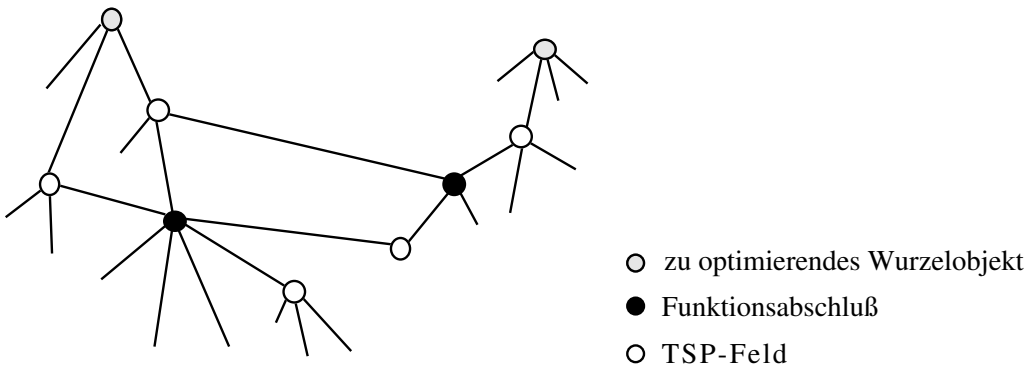



Abbildung 3.3: Dynamischer Optimierungsgraph

pakte persistente Repräsentation des TML-Codes generiert und als linearen PTML-Code (*Persistent TML*) im Objektspeicher abgelegt werden.

Zum dynamischen Zeitpunkt wird der TL-Optimierer von der Programmierumgebung aus mit dem zu optimierenden Objekt aufgerufen. Falls es sich um eine Funktion handelt, wird sie in ihre TML-Darstellung aus dem PTML-Code zurücktransformiert und optimiert. Der Prozeß wird für jedes von dem Ursprungsobjekt aus erreichbare Objekt wiederholt. Damit wird der ganze Aufrufbaum unter dem zu optimierenden Objekt traversiert und dynamisch optimiert (s. Abb. 3.3). Für den Fall eines Moduls (bzw. eines Tupels) bedeutet dies, daß alle darin enthaltenen und transitiv referenzierten Funktionen der Optimierung unterworfen werden. Anschließend wird für alle neu optimierten und nicht vollständig integrierten Funktionen erneut TVM- (bzw. PTML-) Code generiert.

Auf Abb. 3.2 sind die drei Phasen des allgemeinen Modells der dynamischen Optimierung deutlich zu erkennen: PTML-nach-TML-Transformation, Optimierung in der Zwischenrepräsentation (TML) und Code-Neugenerierung. Dabei weist der TL-Optimierer folgende spezifische Merkmale auf, die ihn von den in Abschnitt 3.1 erwähnten Ansätzen unterscheiden:

1. Zum einen weicht die Benutzung einer zweiten Codedarstellung (PTML) von den bisherigen Methoden ab, die neben dem ausführbaren TVM-Bytecode im Objektspeicher gehalten wird, auf der die Rücktransformation in TML beruht. Im Unterschied zum OM-System [Srivastava, Wall 92] wird in Tycoon nicht direkt der Objektcode als Ausgangspunkt für die Reflektion verwendet, sondern eine einfachere, TML-nähere lineare Repräsentation, der PTML-Bytecode. Dem offensichtlichen Nachteil des höheren Speicherbedarfs bei der persistenten Haltung von Funktionsobjekten wird eine recht einfache und eindeutige Umwandlung von TML in die homomorphe PTML-Darstellung und umgekehrt entgegengesetzt. Damit wird der Nachteil der verloren-

gegangenen höhersprachlichen Konstrukte beim OM-System vermieden. Zwar bleibt dadurch die Reflektion nur auf TL beschränkt, im Gegensatz zu OM, wo sich die dynamische Optimierung auf mit unterschiedlichen Übersetzern generierte Objektcodes erstrecken kann. Dieses Verfahren bewahrt jedoch die Maschinenunabhängigkeit, da der PTML- (wie auch der TVM-) Code auf allen Plattformen gleich aussieht. Hier ist ein grundsätzlicher Unterschied in der Philosophie beider Systeme zu erkennen: während OM auf Übersetzerunabhängigkeit der dynamischen Optimierung auf *einer* Maschinenarchitektur absieht, liegen die Bemühungen im Tycoon-System in der globalen Optimierung innerhalb des Systems auf *beliebigen* Plattformen. So kann Tycoon die Information aus den höheren Ebenen besser nutzen, indem es eine geeignete kompakte Repräsentation für TML im Objektspeicher anbietet. Ein nächstes Ziel bei der Systementwicklung von Tycoon ist es jedoch, die PTML- und TVM-Repräsentationen zu verschmelzen, ohne die Vorteile der PTML-Codierung aufzugeben.

2. Das zweite Optimierer-Merkmal ist die destruktive Modifikation der unoptimierten Version einer Funktion im Objektspeicher durch die nach der dynamischen Optimierung entstandene Version, ohne daß ein neues Funktionsobjekt erzeugt wird. D.h., daß im System beide Versionen nicht parallel existieren können. Dadurch wird automatisch die optimierte Funktion allgemein zur Verfügung gestellt.

   Eine andere Alternative wäre die Neuerzeugung des optimierten Funktionsobjekts. Dabei soll ein „Umhängen“ aller von anderen Objekten auf die Funktion gerichteten Referenzen von der alten auf die neue Version vorgenommen werden. Es ist naheliegend, daß der in Tycoon gewählte Ansatz die Konsistenz im System mit weniger Speicher- und Laufzeitbedarf bewahrt. Außerdem ist er der intuitiven Erwartung des Benutzers gerecht, die optimierte Version in allen weiteren Aufrufen zu verwenden.

3. Eine interessante Ausnutzung der Persistenz stellt die dritte Eigenschaft der dynamischen Optimierung in Tycoon dar: die Aufrufbarkeit des TL-Optimierers von der Programmierumgebung aus. Damit ist es dem Benutzer überlassen, welche Funktionen dynamisch optimiert werden sollen.

   Hier taucht das Problem auf, daß dem Benutzer der Zugriff auf den Optimierer als Teil des TL-Übersetzers von dem obersten Sichtbarkeitsbereich (*Tycoon Top Level*) gewährt werden soll. Als ein in TL implementiertes Programm wird der TL-Übersetzer wie alle Tycoon-Objekte im Objektspeicher gehalten. Da er aber ein Dienstprogramm darstellt, das die Evaluationsschleife zum Einlesen, Übersetzen und Ausführen von TL-Ausdrücken durchführt, ist er auf dem obersten Sichtbarkeitsbereicht nicht bekannt. D.h., daß keine Referenz von dem Zustandsvektor mit den definierten Bindungen (*Top Level Vektor*, s. Abschnitt 2.4) zum TL-Übersetzer führt.

   Um trotzdem den Aufruf des TL-Optimierers zu ermöglichen, wird folgendes Verfahren angewendet: Die *Back-End*-Phase des TL-Übersetzers enthält eine Funktion, die die beschriebenen drei Phasen der dynamischen Optimierung auf dem zu optimieren-

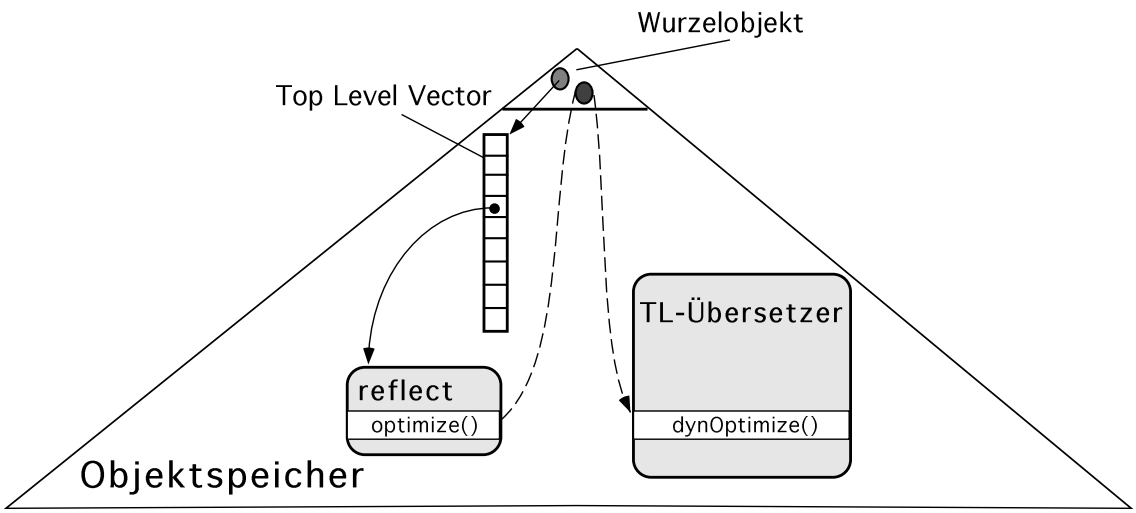



Abbildung 3.4: Indirekter Aufruf der dynamischen Optimierungsfunktion aus dem *Back-End* des TL-Übersetzers

den Objekt durchführt. Während der Übersetzung eines eingegebenen TL-Ausdrucks wird eine Referenz auf diese Funktion an eine bestimmte Stelle im Wurzelobjekt des Objektspeichers abgelegt. Beim Aufruf des dynamischen Optimierers von der Programmierumgebung aus wird dann die gespeicherte Referenz aus dem Wurzelobjekt geholt und das zu optimierende Objekt dem Reflektionszyklus in der *Back-End*-Phase des TL-Übersetzers übergeben.

Abb. 3.4 veranschaulicht den indirekten Zugriff auf das *Back-End* des TL-Übersetzers. Die dynamische Optimierung in Tycoon wird durch den Aufruf der Funktion *optimize* vom Modul *reflect* aus der Tycoon-Standardbibliothek mit dem zu optimierenden Objekt als Argument initiiert. Nach dem Import von *reflect* bekommt der Benutzer über den Zustandsvektor mit den definierten Bindungen (*Top Level Vector*) eine Referenz auf den Modul. Damit kann er die darin enthaltene Funktion *optimize* benutzen. Sie ihrerseits liest die vom Übersetzer im Wurzelobjekt eingetragene Referenz aus und ruft die daran gebundene dynamische Optimierungsfunktion aus dem *Back-End* mit dem zu optimierenden Objekt auf. Diese führt dann darauf die PTML-nach-TML-Transformation, die Optimierung und die erneute Generierung von ausführbarem TVM-Code transitiv durch.

Diese indirekte Verwendung von Übersetzerkomponenten seitens des Benutzers stellt auch eine Art Reflektion dar. Zum einen sind die vom Benutzer eingegebenen Bindungen dem Evaluationszyklus des TL-Übersetzers unterworfen, zum anderen können benutzerdefinierte Bindungen Komponenten des Übersetzers verwenden und insbeson-

dere seinen aktuellen Zustand sehen. Das letzte ist vor allem wegen der Umgebungsvariablen des Systems wichtig, die verschiedene Übersetzeroptionen wie Freiheitsparameter für die Funktionsexpansion, PTML-Generierung oder Zeitmessung darstellen.

## 3.3 Funktionsrepräsentation im Tycoon-Objektspeicher

Aus jeder Funktion erzeugt der TL-Übersetzer ein ausführbares Funktionsobjekt (Funktionsabschluß) und legt es im Objektspeicher ab. Darin sind der TVM- und der PTML-Code sowie auch weitere funktionsrelevante Informationen enthalten. Aus diesen persistenten Strukturen nimmt der dynamische TL-Optimierer den PTML-Code als Ausgangspunkt für den Reflektionszyklus heraus. Darin werden letztendlich die Änderungen nach der erneuten Codegenerierung für die optimierten Funktionsversionen vorgenommen.

Der Funktionsabschluß (*closure*) ist in Tycoon als eine dreistufige Feldstruktur gestaltet (Abb. 3.5). Auf der höchsten Ebene stellt es ein TSP-Feld mit erstem Element *nil* dar. An zweiter Stelle folgt eine Referenz auf den *Literalvektor*. Danach werden die von der Funktion benutzten globalen Variablen aufgeführt. Für eine schon dynamisch optimierte Funktion gibt es an dieser Stelle keine Globalen: sie sind bereits propagiert und im Code eingesetzt worden. Da die neue Version dasselbe Speicherobjekt der unoptimierten Funktion verwendet und modifiziert, werden darin anstelle der Globalen *nil*-Werte eingetragen.

Der Literalvektor enthält die vom TVM-Bytecode benutzten konstanten Objekte, die länger als ein Speicherwort sind (Tupel, Felder, Zeichenketten, reelle Zahlen, etc.). An der ersten Position befindet sich eine Referenz zum ausführbaren TVM-Bytecode, der durch ein byteorientiertes Feld repräsentiert wird. An zweiter Position liegt die Referenz zur PTML-Struktur, wo bestimmte Optimiererinformationen, der PTML-Bytecode und die Variablennamen (zur Wiedererkennung der benutzten TL-Variablen, für Debuggingzwecke) gehalten werden. Ähnlich wie der TVM-Code wird auch der PTML-Code als byteorientiertes Feld dargestellt (s. Abschnitt 2.4).

Folgende Informationen werden zum Zweck der dynamischen Optimierung im Funktionsobjekt neben dem PTML-Code gespeichert:

- Das Feld *isRecFun* bestimmt, ob die Funktion rekursiv ist. Dieses ist notwendig, weil das TML-Konstrukt Y zur Definition rekursiver Funktionen (s. Abschnitt 4.3.2) beim Aufruf einer der beteiligten Abstraktionen von außerhalb nicht erkennbar ist.
- Das Feld *unknownRefs* enthält die Anzahl der innerhalb des Funktionsrumpfes benutzten globalen Variablen. Daran kann der Optimierer erkennen, ob eine Funktion einer dynamischen Optimierung bedarf (s. Abschnitt 5.4). Die Bedingung dazu ist $unknownRefs \neq 0$, d.h. daß alle globalen Definitionen im Code integriert sind.
- Das Feld *cost* enthält die Kosteninformation über den Funktionsrumpf, die als ein Maß für die Codegröße in der Expansionsheuristik verwendet wird. Nähere Erläuterungen

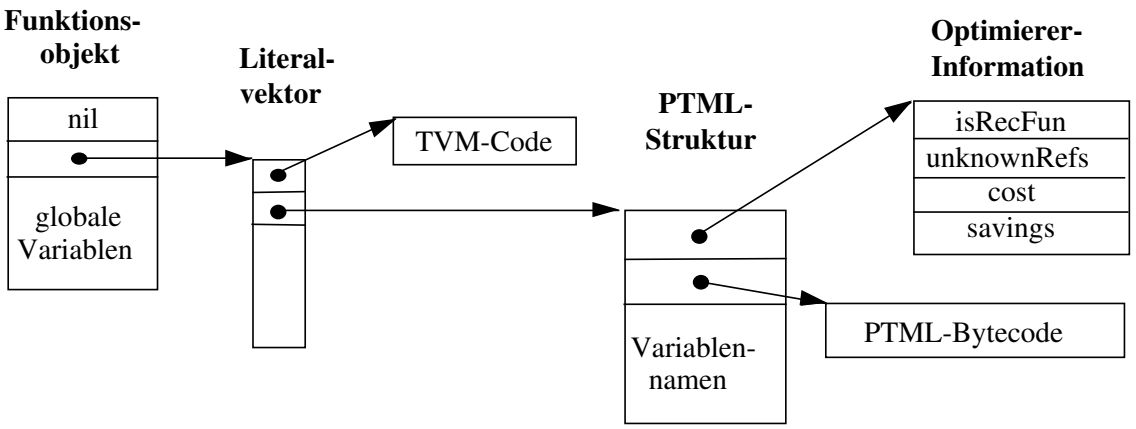



Abbildung 3.5: Struktur eines Funktionsobjekts im Objektspeicher

über die Kostenberechnung sind in Abschnitt 5.3.2.1 zu finden.

- Das Feld *savings* erfaßt die indirekten Auswirkungen einer Funktionsexpansion bei einer anschließenden Reduktion des substituierten Aufrufs, indem für jeden Parameter der Funktion die von ihm verursachten potentiellen Kostenminderungen bei seiner Bindung an ein konstantes Argument geschätzt werden. Die Heuristik zur *Savings*-Berechnung wird in Abschnitt 5.3.2.1 ausführlich diskutiert.

Hier wird besonders der Vorteil der Ausnutzung der Persistenz zum Zweck der Effizienzsteigerung des Optimierers selbst deutlich. Statt die Information über die aus PTML wiederhergestellte TML-Funktionsabstraktion erneut durch Traversieren des Rumpfes zu gewinnen, wird sie zur weiteren Verwendung im Funktionsobjekt bereitgestellt. Aufgabe des Optimierers ist es, während der Transformationen für ihre Aktualisierung zu sorgen. Die Informationsfelder beziehen sich auf die ganze Funktion und dienen dazu, eine überflüssige Traversierung zwecks Kosten- bzw. *Savings*-Berechnung für eine Expansionsentscheidung zu vermeiden. Außerdem kann der Optimierer sofort erkennen, ob ihm eine unoptimierte (bzw. nur statisch optimierte) oder eine dynamisch optimierte Funktion vorliegt, um entsprechend eine geschachtelte Optimierung vorzunehmen oder nicht (vgl. Abschnitt 5.3.2.2).

# 4 Die CPS-basierte Sprache TML

Während der Übersetzung durchlaufen die höhersprachlichen TL-Konstrukte verschiedene Zwischenrepräsentationen, die für die jeweiligen Übersetzungsphasen geeignet sind. So benutzt der Typüberprüfer im *Front-End* den abstrakten TL-Syntaxbaum, der vom TL-Parser aus dem TL-Quelltext erstellt wird und eine bequeme Umgebung für die Typanalyse darstellt. Weitere Repräsentationen sind die im Kapitel 2 erwähnten TVM- und PTML-Bytecodes, die der effizienten Interpretation bzw. der reflektiven Transformation des Programmcodes dienen. Für die Optimierung spielt die TML-Zwischenrepräsentation mit ihrer Einfachheit, Generalität und Ausdrucksmächtigkeit die zentrale Rolle.

Dieses Kapitel ist der Beschreibung von TML gewidmet. In Abschnitt 4.1 wird ein Überblick über existierende CPS-Sprachen und ihre Verwendung in optimierenden Übersetzern gegeben, wobei auch auf ihre Unterschiede zu TML eingegangen wird. Die Syntax und Semantik von TML werden in Abschnitt 4.2 diskutiert. Anschließend wird in Abschnitt 4.3 die Überführung verschiedener für die Optimierung interessanter TL-Konstrukte nach TML geschildert. Hier wird deutlich werden, welche Komplexitätsreduktion eine solche Sprachrepräsentation für die Optimierung und Codegenerierung mit sich bringt.

## 4.1 Charakterisierung und Überblick

Ziele wie Vereinfachung und Generalisierung von Programmübersetzung, -transformation und -analyse lassen sich sehr gut durch Darstellung des Programmcodes in *Continuation Passing Style* (*CPS*) erreichen. Aufbauend auf dem Lambda-Kalkül integriert CPS die Vorteile dieser formalen funktionalen Programmform und findet in modernen Übersetzern breite Verwendung.

Zum einen ist eine Lambda-Kalkül-Repräsentation eine sehr einfache Sprache mit wenigen Konstrukten und klarer Semantik. Dadurch eignet sie sich besonders für die Codeanalyse und -transformation und erleichtert vor allem Korrektheitsbeweise für die Implementierung. Zum anderen ist der Lambda-Kalkül Turing-vollständig. Wenn also eine Implementation für den Lambda-Kalkül existiert, kann man dadurch alle Sprachen implementieren, indem sie in die Lambda-Kalkül-Repräsentation transformiert werden ([Peyton-Jones 87]). Durch ihren

funktionalen Charakter ist diese Repräsentation insbesondere für funktionale Sprachen angemessen. Imperative Konstrukte, wie Zuweisungen, Schleifen und Ausnahmebehandlung lassen sich durch geeignete Transformationen und elementare Operationen (s. Abschnitt 4.3) abbilden.

Eine CPS-Repräsentation unterscheidet sich vom allgemeinen Lambda-Kalkül in zwei Aspekten:

1. CPS-Funktionen liefern keinen Wert zurück. Sie erhalten vielmehr beim Aufruf ein zusätzliches Argument – die Fortsetzung (*continuation*) – das eine Funktion mit einem an das Funktionsergebnis zu bindenden Parameter sein muß. Diese Fortsetzungsfunktion bestimmt den weiteren Programmlauf und stellt eine explizite parametrisierte Angabe des Kontrollflusses durch einen Funktionsaufruf dar. Damit geht der höhersprachliche implizite *return*-Charakter der Funktionen in eine explizite Kontrollflußdarstellung der *goto*-Form. Sie nähert sich sehr an den linearisierten Maschinencode und vereinfacht deswegen die Codegenerierung. Die Erübrigung einer Kontrollflußanalyse ist jedoch der größte Vorteil der CPS-Zwischensprachen, da sie Voraussetzung für alle Optimierungstransformationen ist.

2. TML unterstützt das Verhalten einer Programmiersprache mit strikter Evaluation der Funktionsargumente (wie ML, Pascal, Common-Lisp, TL), was beim Lambda-Kalkül nicht festgelegt ist. Dies wird in TML realisiert, indem nicht-triviale Argumente in Funktionsapplikationen nicht erlaubt sind. D.h. Argumente einer Funktionsapplikation dürfen nur Konstanten, Variablen und Lambda-Abstraktionen, nicht aber wiederum Applikationen sein, wodurch die Evaluation der Argumente vor ihrer Bindung an die formalen Parameter garantiert ist. Diese TML-Eigenschaft bedingt die problemlose Substitution der formalen Parametern mit Argumenten in einem Funktionsrumpf ohne die Gefahr mehrfacher Argumentevaluation oder Verletzung von Seiteneffekten, wie es der Fall in einer allgemeinen Lambda-Kalkül-Repräsentation wäre.

Eine spezifische Eigenschaft von TML ist, daß bereits bei der Transformation der TL-Konstrukte in TML während der *Front-End*-Phase des Übersetzers jede Variable eindeutig identifiziert wird. Eine entsprechende $\alpha$-Konvertierung[1] wird automatisch bei der Generierung des TML-Codes durchgeführt. Dadurch werden potentielle Namenskonflikte bei Variablensubstitutionen, die zur Erweiterung von Sichtbarkeitbereichen führen, ausgeschlossen. Der aus der TL-nach-TML-Konvertierung resultierende TML-Baum ist direkt für Analyse und Optimierungen geeignet.

Wie schon erwähnt wurde, verwenden mehrere moderne optimierende Übersetzer die CPS-Coderepräsentation. Es gibt jedoch einige Differenzen in Bezug auf die einzelnen CPS-Dialekte und ihre Verwendung in den verschiedenen Übersetzungsphasen.

[1] Mit $\alpha$-Konvertierung wird genau der Prozeß bezeichnet, bei dem Namenskonflikte für Variablen (auch aus verschiedenen Sichtbarkeitsbereichen) ausgeschlossen werden.

Zuerst wurde die CPS-Repräsentation von Steele in *Rabbit* eingesetzt, einem optimierenden Übersetzer für Scheme ([Steele 78]). Die funktionale Natur von CPS läßt ihre saubere, elegante und wohl verstandene Behandlung und Interpretation als eine Teilmenge von Scheme zu. Steele nutzt jedoch die Vorteile der daraus resultierenden Komplexitätsreduktion nur bei der Codegenerierung, führt jedoch alle Optimierungstransformationen auf der Quellsprachebene durch.

In [Kranz et al. 86] wird ein weiterer CPS-basierter optimierender Übersetzer für Scheme, *Orbit*, präsentiert. Er zeichnet sich durch die Durchführung einer Reihe von Optimierungstransformationen in der CPS-Form sowie durch eine ausführliche Analyse für die Speicherallokation von den aus Funktionsabstraktionen resultierenden ausführbaren Funktionsabschlüssen (*closures*) aus. Die Autoren erzielen sehr gute Ergebnisse bei der Optimierung von Scheme-Code und demonstrieren damit die Konkurrenzfähigkeit von modernen Lisp-Dialekten zu den allgemein als effizient anerkannten Sprachen der Algol-Familie. In dieser Hinsicht existiert noch ein sehr ähnlicher Scheme-Übersetzer, HARE, beschrieben in [Teodosiu 91].

*Standard ML of New Jersey (SML)* ist der in [Appel 89] und [Appel 92] vorgestellte CPS-basierte optimierende Übersetzer für ML. In SML wird die CPS-Repräsentation für codeverbessernden Transformationen wie Reduktion, Funktionsexpansion, Code-Verschiebung (*hoisting*) und Entfernen gemeinsamer Teilausdrücke verwendet, bei denen interessante Analyseheuristiken verwendet werden. In Bezug auf die Syntax beruht SML jedoch auf einem größeren Satz an Sprachkonstrukten im Vergleich zu TML.

Kelsey ([Kelsey 89], [Kelsey, Hudak 89]) erweitert das CPS-Konzept auch auf nicht funktionale Sprachen, indem er einen einfachen universellen Übersetzer vorstellt, der auf der Grundlage einer CPS-basierten Zwischenrepräsentation die Optimierungstransformationen, dann eine Codeanpassung an die Zielmaschinenkonfiguration und anschließend die Evaluation durchführt. Nach einer vorausgehenden Transformation der jeweiligen Quellsprache in die CPS-Form findet die quellsprachenunabhängige Weiterübersetzung statt. Kelsey demonstriert das Verfahren an Pascal, Basic und Scheme. TML unterscheidet sich von Kelsey's CPS-Dialekt durch die stärker ausgeprägte Reduzierung der Sprachkonstrukte (z.B. wird in TML der *return*-Operator durch einen Fortsetzungsaufruf abgebildet).

[Gawecki 91] stellt einen weiteren Anwendungsbereich einer CPS-Sprache durch ihren Einsatz in einem optimierenden Übersetzer für Smalltalk vor. Wie es in objekt-orientierten Sprachen üblich ist, konzentriert sich dieser Ansatz besonders auf die Elimination von Botschaften durch ihre Substitution mit Prozeduraufrufen. Diese können dann auf der Grundlage einer Typanalyse expandiert und reduziert werden.

Die beiden letzten Beiträge zeigen, daß sich die CPS-Form trotz ihrer funktionalen Grundlage auf keinen Fall ausschließlich für funktionale Sprachen eignet, sondern eine einheitliche und effiziente Programmdarstellung für Programmiersprachen unterschiedlicher Paradigmen anbietet. TML unterstützt diese These, indem sie eine polymorphe Sprache mit Funktionen

höherer Ordnung und imperativen Elementen vollständig implementiert und einen wesentlichen Beitrag für die Realisierung verschiedener Transformationen in Bezug auf Komplexitätsreduktion, Generalisierung und Korrektheitsüberprüfung leistet.

Die Zwischensprache TML ist auf der Grundlage der erwähnten Übersetzerimplementationen entstanden und unterscheidet sich von den dort verwendeten CPS-Dialekten in zwei Hinsichten:

- Zum einen ist die Anzahl der Sprachkonstrukte sehr stark reduziert – insgesamt 6 Sprachelemente, wie aus der abstrakten TML-Syntax (s.u.) ersichtlich ist. Vielmehr wird die Funktionalität von den verschiedenen erweiterbaren elementaren Operationen getragen, die sowohl für Speicherzugriffe, arithmetische und boolesche Operationen als auch für die Definition von rekursiven Funktionen (anders als z.B. bei [Appel 92], wo es ein extra Sprachkonstrukt *FIX* für diesen Zweck gibt) verantwortlich sind.
- Zum anderen erhält jede Funktion in TML eine speziell gekennzeichnete Fortsetzung – die Ausnahmefortsetzung (*exception continuation*) – die den Kontrollfluß im Ausnahmefall spezifiziert und ein Ausnahmepaket an die jeweilige Prozedur zur Ausnahmebehandlung (*handler*) übergibt. Dadurch entfällt die Notwendigkeit zur speziellen Behandlung von Ausnahmen (*exceptions*).

## 4.2 Syntax- und Semantikbeschreibung von TML

Die in diesem Abschnitt enthaltene TML-Beschreibung lehnt sich an [Gawecki, Matthes 94] an. Es werden die für die Optimierung relevanten Merkmale der CPS-orientierte Zwischensprache TML diskutiert. Für Details wird auf den zitierten Bericht verwiesen.

### 4.2.1 Abstrakte Syntax von TML

Die abstrakte Syntax von TML ist in Abb. 4.1 angegeben. Das Minimum an syntaktischen Konstrukten ist eine wesentliche Eigenschaft der Zwischensprache. Es existieren vier Kategorien: Variablen, Werte, Funktionsabstraktionen und Funktionsapplikationen. In TML wird zwischen Wert- und Fortsetzungsidentifikatoren differenziert. Die ersteren bezeichnen „normale“ Werte, während die letzteren auf der Maschinenebene Sprungmarken (*labels*) entsprechen. Obwohl beide Arten von Wertidentifikatoren dieselbe Semantik und interne Darstellung (s. Anhang A) haben, deuten sie auf die unterschiedliche Behandlung der benutzerdefinierten Funktionen und der vom Übersetzer definierten Fortsetzungen durch den Codegenerator hin.

Fortsetzungen entsprechen auf der Maschinenebene bedingten bzw. unbedingten Sprüngen. Sie können höchstens einen Parameter haben, der seinerseits keine Fortsetzung sein darf. Fortsetzungen stellen in TML keine Werte erster Ordnung dar und werden nicht wie benutzerdefinierte Funktionen in Datenstrukturen (als Funktionsabschlüsse) abgelegt, woher sie

$l \in Lit$ *Literalkonstanten (Objektreferenzen)*
$t \in Temp$ *temporäre Variablen*
$c \in Cont$ *Fortsetzungsvariablen*
$v \in Var$ *Variablen (Identifikatoren)*
$p \in Prim$ *elementare Operationen*

$val \in Val$ *Werte*
$abs \in Abs$ *Abstraktionen*
$app \in App$ *Applikationen*

$$
\begin{array}{lcll}
v & ::= & t \mid c & \\
val & ::= & l \mid v \mid abs & \\
abs & ::= & \lambda(v_1 \ldots v_n) app & n \geq 0 \\
app & ::= & (val_0\ val_1 \ldots val_n) & n \geq 0 \\
 & \mid & (p\ val_1 \ldots val_n) & n \geq 0
\end{array}
$$

Abbildung 4.1: Abstrakte Syntax von TML (aus [Gawecki, Matthes 94])

zu ihrer Anwendung erstmals geholt werden müßten. Diese Einschränkung erlaubt die Implementation von TML-Prozeduraufrufen durch effiziente stapelbasierte Verwaltungsstrukturen auf konventionellen Hardware-Architekturen.

Eine typische Eigenschaft von TML ist, daß die benutzerdefinierten Prozeduren neben ihren Wertparametern immer zwei Fortsetzungsparameter erhalten. Sie werden beim Aufruf an die Programmfortsetzungen für den Normalfall und für den Ausnahmefall gebunden. Die Ausnahmefortsetzung steht immer an der vorletzten Position in der Parameterliste, die normale Fortsetzung an der letzten. Dadurch wird in TML neben dem normalen Kontrollfluß des Programms auch der Kontrollfluß im Ausnahmefall explizit gekennzeichnet, was einen wichtigen Beitrag zur uniformen Optimierung leistet.

Zum Zweck der besseren Lesbarkeit des in der Arbeit präsentierten TML-Codes werden benutzerdefinierte Funktionen (**proc**-Abstraktionen) und Fortsetzungen (**cont**-Abstraktionen) explizit bezeichnet. Die folgenden syntaktischen Äquivalenzen stellen Spezialisierungen der Kategorie der TML-Abstraktionen mit den aufgeführten Restriktionen dar:

$$
\begin{array}{lcll}
\textbf{proc}(t_1\ \ldots\ t_n\ c_e\ c_c)\ app & \equiv & \lambda(t_1\ \ldots\ t_n\ c_e\ c_c)\ app & n\ \geq\ 0 \\
\textbf{cont}(t) app & \equiv & \lambda(t)\ app & \\
\textbf{cont}() app & \equiv & \lambda()\ app &
\end{array}
$$

In der Kategorie der TML-Applikationen sind an Funktionsposition ($val_0$ in Abb. 4.1) folgende vier Alternativen erlaubt:

- eine Variable, die an eine Funktionsabstraktion gebunden sein muß
- eine Objektreferenz (*oid*) auf ein gespeichertes Funktionsobjekt
- eine Lambda-Abstraktion
- eine elementare Operation

Die elementaren Operationen implementieren die Grundfunktionalität der Quellsprache: sie führen Speicherzugriffe sowie boolesche und arithmetische Operationen durch, sind aber auch für die Ausnahmebehandlung, für die Definition rekursiver Funktionen und für andere spezifische Aufgaben verantwortlich. Darauf wird in Abschnitt 4.2.2 genauer eingegangen.

Nicht zuletzt ist die Einschränkung der Argumentausprägungen in Funktionsaufrufen von Bedeutung. Sie dürfen nur direkte Werte, d.h. Konstanten (Literale), Variablen oder Lambda-Abstraktionen, sein. Nicht-triviale Argumente wie Funktionsaufrufe sind nicht erlaubt. Diese typische CPS-Eigenschaft spielt eine wesentliche Rolle bei den Optimierungstransformationen, da bei der Funktionsexpansion bzw. Beta-Reduktion die Parametersubstitution ohne Seiteneffektanalyse auskommen kann. (Die Gefahr einer Mehrfachevaluation der Argumente wird durch die TML-Syntax abgewendet.)

### 4.2.2 Elementare Operationen

Die minimale Syntax von TML basiert auf der Verlagerung der Quellsprachenfunktionalität auf die elementaren Operationen. Die Menge der vordefinierten elementaren Operationen ist in Abb. 4.2 aufgelistet. Per Definition wird die elementare Operation bei einem Aufruf auf die Argumente evaluiert, genau eine der Fortsetzungen ausgewählt und mit dem Ergebnis der Auswertung (falls ein solches existiert) angewendet. Die elementaren arithmetischen Operationen erhalten z.B. zwei Fortsetzungen: eine $normaleFortsetzung$ mit dem berechneten Ergebnis als Argument und eine $Ausnahmefortsetzung$ für den Fall eines Überlaufs.

Die Anzahl der vordefinierten elementaren Operationen ist zwar klein, aus pragmatischen Gründen aber nicht minimal. Die Existenz der beiden Operationen *array* und *vector* nebeneinander wird z.B. durch die Fähigkeit des Optimierers gerechtfertigt, die Zugriffe auf die unveränderlichen Feldelemente aus einer mit *vector* erzeugten Speicherstruktur zu eliminieren und ihre konstanten Inhalte im Code einzusetzen. Darüberhinaus kann auch der Objektspeicher davon profitieren, indem er in Abhängigkeit der Mutabilität der Objekte verschiedene Blockierungsmechanismen einsetzt.

Im nach dem *add-on*-Ansatz [Matthes, Schmidt 91] implementierten Tycoon-System ist es durchaus möglich neue elementare Operationen zu definieren, die der Realisierung spezialisierer Datenmodelle mit eingebauten komplexen Instruktionen (z.B. aus der objekt-orientierten Welt) dienen. Dazu ist es notwendig, für jede neu eingeführte elementare Operation folgende zusätzliche Funktionen anzugeben:

| | |
|---|---|
| $(p\ x\ y\ c_e\ c_c)$ | ganzzahlige Arithmetik, $p \in \{+, -, *, /, \%\}$ |
| $(p\ x\ y\ c_1\ c_2)$ | ganzzahlige Vergleiche, $p \in \{<, >, <=, >=\}$ |
| $(p\ x\ y\ c)$ | Bit-Operationen, $p \in \{<<, >>, \&, \mid, \hat{}, \tilde{}\}$ |
| (char2int $x\ c$) | Konvertierung eines Zeichens in eine ganze Zahl |
| (int2char $x\ c$) | Konvertierung einer ganzen Zahl in ein Zeichen |
| (array $x_1 \ldots x_n\ c$) | Erzeugung eines veränderbaren Feldes mit $n$ Speicherwörtern |
| (vector $x_1 \ldots x_n\ c$) | Erzeugung eines unveränderbaren Feldes |
| (new $n$ *init* $c$) | Erzeugung eines veränderbaren Feldes mit $n$ Speicherwörtern, die mit *init* initialisiert werden |
| ($new $n$ *init* $c$) | Erzeugung eines veränderbaren Feldes mit $n$ einfachen Byte-Werten, die mit *init* initialisiert werden |
| ([ ] $x\ i\ c$) | Feldzugriff: indirekt indiziertes Laden |
| ([ ]:= $x\ i\ v\ c$) | Feldmodifikation: indirekt indiziertes Speichern |
| ($[ ] $x\ i\ c$) | Byte-Feld-Zugriff |
| ($[ ]:= $x\ i\ v\ c$) | Byte-Feld-Modifikation |
| (== $x$<br>$tag_1 \ldots tag_n$<br>$c_1 \ldots c_n\ [c_{n+1}]$) | auf Objektidentität basierte Fallanalyse mit Werten und den jeweiligen Verzweigungen (else-Zweig optional) |
| $(Y\ \lambda(c\ f_1 \ldots f_n)(c\ C\ F_1 \ldots F_n))$ | der $Y$-Kombinator (s. § 4.3.2) |
| (size *arr* $c$) | Größe eines Feldes bzw. Byte-Feldes |
| (move $n$ *src srcOffset dst dstOffset* $c$) | Kopieren von Feldbereichen |
| ($move $n$ *src srcOffset dst dstOffset* $c$) | Kopieren von Byte-Feld-Bereichen |
| (ccall *fmt cfn* $c_1\ c_2$) | Aufruf einer externen C-Funktion |
| (pushHandler $c_1\ c_2$) | Installation von $c_1$ als Ausnahmehandler |
| (popHandler $c$) | Entfernung des obersten Ausnahmehandlers |
| (raise $x$) | Auslösen der Ausnahme $x$ |

Abbildung 4.2: Elementare Operationen (aus [Gawecki, Matthes 94])

- Eine Generierungsfunktion, die Zielmaschinencode für einen gegebenen Aufruf der jeweiligen elementaren Operation erzeugt. Sie wandelt Aufrufe der elementaren TML-Operationen in eine Sequenz von Instruktionen für die entsprechende Zielmaschinenarchitektur um.
- Eine (Meta-)Evaluationsfunktion, die vom Optimierer zur Auswertung konstanter Ausdrücke (*constant folding*) eingesetzt wird. Ein Beispiel ist der Aufruf der elementaren Operation '+' mit konstanten Wertargumenten:

  *(+ 1 2 e c)*

  Dieser Aufruf wird in der Reduktionsphase des Optimierers durch die Evaluationsfunktion der Operation '+' ausgewertet und durch die Anwendung der normalen Fortsetzung auf das Resultat ersetzt:

  *(c 3)*

  Dabei müssen eventuell auch Seiteneffektbedingungen berücksichtigt werden, weil die Quellsprachenfunktionalität und damit auch die Seiteneffekte in den elementaren Operationen stecken. Beispiele für Operationen mit Seiteneffekten sind '[]:=' und '$[]:=', die ein Element von einem wortorientierten bzw. byteorientierten TSP-Feld modifizieren.

### 4.2.3 Ausnahmen

Wie schon erläutert wurde, beruht die Ausnahmebehandlug in TML auf einem explizit für jede benutzerdefinierte Funktion angegebenen Fortsetzungsparameter. Beim Funktionsaufruf wird eine Ausnahmefortsetzung an diesen Parameter gebunden, womit ihre Abstraktion als Ausnahmebehandlung (*handler*) für die jeweilige Funktion installiert wird. Durch diese Parameterübergabe werden die Prozeduren zur Ausnahmebehandlung in der lexikalischen Umgebung über die Aufrufhierarchie verwaltet. Durch die explizite Bezeichnung des Ausnahmekontrollflusses kann der Optimierer Ausnahmen zusammen mit den normalen Fortsetzungen uniform behandeln.

Vom Gesichtspunkt der Implementierung ist jedoch eine Stapelverwaltung der Prozeduren zur Ausnahmebehandlung effizienter. Aus diesem Grund werden vor der Codegenerierung die Ausnahmefortsetzungsargumente aus den Funktionsaufrufen entfernt und eine geeignete Sequenz der elementaren Operationen *pushHandler* und *popHandler* zur Verwaltung eines Ausnahmelaufzeitstapels in den Code einfügt. Diese Aufgabe wird von der Ausnahme-Konvertierung übernommen.

Durch dieses Verfahren wird einerseits der Optimierer bzw. der Codegenerator von Details der Ausnahmeverwaltung verschont und andererseits eine effiziente Ausnahmebehandlung zur Laufzeit erreicht.

### 4.2.4 Das Back-End des TL-Übersetzers

Im folgenden werden die einzelnen Phasen des *Back-Ends* des TL-Übersetzers vorgestellt:

1. **TL-nach-TML-Transformation**: Umwandlung des typisierten TL-Syntaxbaumes aus der *Front-End*-Phase in einen untypisierten TML-Baum. Hier wird auch eine $\alpha$-Konvertierung durchgeführt, um Namenskonflikte bei Optimierungstransformationen zu vermeiden.
2. **TML-Optimierung** (optional): die in dieser Arbeit beschriebenen codeverbessernden Transformationen auf dem TML-Baum.
3. **Ausnahme-Konvertierung**: die schon erwähnte Eliminierung der Ausnahmefortsetzungen aus den Funktionsabstraktionen und -applikationen. Analog zur Optimierung wird auch hier der TML-Baum bearbeitet.
4. **Zielmaschinencodegenerierung**: Umgebungsanalyse, Erzeugung von Funktionsabschlüssen, Generierung vom TVM- und (optional) PTML-Code sowie Ablage der Funktionsabschlüsse im Objektspeicher.

Bei der dynamischen Optimierung (s. Abschnitt 3.2) wird die *Back-End*-Phase des TL-Übersetzers von der Programmierumgebung aus durch den Benutzer aufgerufen. Nach der Rücktransformation von PTML in TML geht es mit Phase 2 (TML-Optimierung) weiter.

## 4.3 Abbildung höhersprachlicher Konstrukte auf TML

Im folgenden wird die Überführung der für die Optimierung interessanten TL-Konstrukte in TML-Form dargestellt. Das Ziel ist, auf der einen Seite die Ausdrucksmächtigkeit von TML zu demonstrieren, auf der anderen Seite auf die vereinfachten und einheitlichen Strukturen hinzuweisen, auf denen der Optimierer codeverbessernde Transformationen durchführt.

Jeder TL-Ausdruck erhält bei der TL-nach-TML-Transformation zwei aktuelle Fortsetzungen: die normale und die Ausnahmefortsetzung. Diese werden zunächst in der initialen Umgebung des Tycoon-Systems durch die Evaluationsschleife definiert, die einen TL-Ausdruck einliest, übersetzt, auswertet und das Ergebnis ausgibt. Diese initialen Fortsetzungen werden im folgenden als *cc* (für die normale) und *ce* (für die Ausnahmefortsetzung) bezeichnet. Die erste bekommt das Ergebnis der Evaluation des TL-Ausdrucks und gibt es aus, die zweite gibt im Ausnahmefall Informationen über die ausgelöste (und nicht abgefangene) Ausnahme sowie den Laufzeitstapel aus.

In Anlehnung an [Gawecki, Matthes 94] wird für die folgenden Beispiele eine Transformationsfunktion eingeführt:

$\ll A, ce, cc \gg$ *TML-Baum*

Sie bekommt als Argument den zu transformierenden TL-Ausdruck mit den beiden aktuellen Fortsetzungen und erzeugt einen TML-Baum, der wiederum Aufrufe an die Transformationsfunktion enthalten kann.

### 4.3.1 Konstanten, Ausdrücke, Bindungen

Konstanten wie Zahlen und Zeichenketten evaluieren sich selbst und werden in TML durch Aufruf der normalen Fortsetzung mit der Konstante als Argument ausgedrückt:

```
≪13, ce, cc≫    (cc 13)
```

Komplizierte Ausdrücke werden zuerst in elementare Operationen zerlegt. Die Zwischenergebnisse werden an temporäre Variablen gebunden, worauf dann die nächsten Operationen angewendet werden. Dadurch wird die genaue Reihenfolge der TL-Auswertung in TML abgebildet[2]:

```
≪1 + {2 * 3 } + 4, ce, cc≫    (*_72 2 3 ce cont(t_1)
                               (+_50 1 t_1 ce cont(t_2)
                               (+_50 t_2 4 ce cont(t_3)
                               (cc t_3))))
```

Es ist darauf hinzuweisen, daß *+_50* und **_72* globale Bezeichner darstellen, die bestimmte in der Programmierumgebung bereits definierte Funktionen referenzieren.

Sequentielle und parallele Wertbindungen werden in geschachtelte Abstraktionen übersetzt, wobei jeder Wert an einen TML-Bezeichner gebunden wird, der dem TL-Namen entspricht. Der Wert des letzten Ausdrucks der Sequenz wird als Ergebnis zurückgegeben, d.h. in CPS-Terminologie die *cc*-Fortsetzung wird aufgerufen:

```
≪let a = A let b = B, ce, cc≫    ≪A, ce, cont(a_1)    Bindung von a an A
                                 ≪B, ce, cont(b_2)    Bindung von b an B
                                 (cc b_2)≫≫           Rückgabe von b
```

Die TL-Sichbarkeitsregeln für parallele Bindungen werden durch die $\alpha$-konvertierten TML-Bezeichner explizit gemacht:

```
≪let a = A     ≪A, ce, cont(a_1)    Berechnung der ersten Variable a
let a = B      ≪B, ce, cont(a_2)    Berechnung der zweiten Variable a
and b = a,     (λ(b_3)              Bindung von b ...
ce,              (cc b_3)             Rückgabe von b
cc≫            a_1)≫≫               ... an die erste Variable a
```

[2]Die Zahlen, die in den TML-Bezeichner enthalten sind, dienen der besseren Lesbarkeit, um gleichnamige TL-Variablen auf der TML-Ebene zu unterscheiden. Sie werden folgendermaßen zusammengesetzt: *xxx_nn*. Die eindeutige interne Unterscheidung zwischen TML-Variablenknoten wird durch die Objektidentität garantiert.

### 4.3.2 Funktionen

Jede TL-Funktion erhält auf der TML-Ebene zwei zusätzliche formale Parameter für die beiden möglichen Fortsetzungen bei einem Funktionsaufruf – Ausnahmebehandlung bzw. Rückgabe des Resultats:

```
≪fun(x, y :Int) S, ce, cc≫    (cc proc(x_6 y_7 e_4 c_5) ≪S, e_4, c_5≫)
```

Dabei werden die Fortsetzungsparameter der TML-Abstraktion als Argumente der Transformationsfunktion übergeben, die die Rolle der funktionsspezifischen Ausnahmebehandlung (*e_4*) und normale Ergebnisrückgabe (*c_5*) erfüllen.

Wechselseitig rekursive Funktionsbindungen werden durch die elementare Operation Y in TML eingeführt, welche eine CPS-Version des Fixpunktkombinators aus dem Lambda-Kalkül ist. Diese Operation bekommt als Argument eine Lambda-Abstraktion, deren kleinster Fixpunkt eben ein Vektor der wechselseitig rekursiven Funktionen ist. An erster Position in diesem Vektor muß eine parameterlose Fortsetzungsabstraktion stehen, die den weiteren Programmablauf darstellt. Im folgenden Beispiel werden zwei wechselseitig rekursive Funktionen definiert und an die Namen *f* und *g* gebunden; anschließend wird die zweite (*g*) aufgerufen:

```
≪let rec f(a :Int) :Int = F   (Y proc(c_14 g_5 f_4)
and g(b :Int) :Int = G         (c_14
g(7),                           cont() (g_5 7 cc)                    Aufruf von g
ce,                             proc(b_8 e_6 c_7) ≪G, e_6, c_7≫      die Funktion g
cc≫                             proc(a_12 e_10 c_11) ≪F, e_6, c_7≫))  die Funktion f
```

Mit demselben *Y*-Konstrukt werden auch die TL-Schleifen (**for**, **loop** und **while**) auf der TML-Ebene repräsentiert. Sie werden auf restrekursive Funktionen abgebildet und erlauben damit die einheitliche Behandlung von Schleifen und rekursiven Funktionen. Das folgende Beispiel gibt den TML-Code für eine **for**-Schleife wieder:

```
≪for i = 1 upto 10 do   (λ(forEnd_1)
  F                      (Y proc(c_2 for_3)
end,                     (c_2
ce,                       cont() (for_3 1)                  Beginn der Schleife
cc≫                       cont(i_4)                         rekursive Schleifenfunktion
                            (> i_4 10 forEnd_1 cont()       Schleifenausgang
                            ≪F, ce, cc≫                     Schleifenkörper
                            (+ i_4 1 cont(t_5)              Iterationsschritt
                            (for_3 t_5)))))                 nächste Iteration
                          cont()
                           (cc ok))
```

### 4.3.3 Wertkonstuktoren

Aggregierende Wertkonstruktoren werden in entsprechende Aufrufe elementarer Operationen übersetzt, welche die jeweiligen Speicherstrukturen (*arrays*) allozieren und die Elementwerte initialisieren.

Parallel definierte rekursive Bindungen werden in 4 Schritten in TML durchgeführt:

1. Allokation von Aggregaten
2. Initialisierung von Variablen für die einfachen Bindungen
3. Bindung der rekursiven Funktionen
4. Initialisierung der Aggregate

Hierzu ein Beispiel:

```
≪let rec tup :T =                (new 2 nil cont(tup_4)          1. Aggregatallokation
  tuple f end                    (h_1 99 ce cont(x_6)            2. Initialisierung von x
and f() :T = g(x f)              (Y proc(c_15 g_7 f_5)           3. Bindung der rekursiven
and x :Int = h(99)                                                  Funktionsvariablen
and g(a :Int b :Fun():T) :T      (c_15 cont()
   = tup,                         ([]:= tup_4 0 1 cont()        4. Initialisierung von tup
ce,                               ([]:= tup_4 1 f_5 cont()
cc≫                               (cc g_7)))
                                 proc(x_10 b_11 e_8 c_9)
                                  (c_9 tup_4)
                                 proc(e_12 c_13)
                                  (g_7 x_6 f_5 e_12 c_13)))))
```

Zu beachten ist, daß ein Tupel durch ein TSP-Feld dargestellt wird, an dessen erster Position eine Zahl für die Tupelvariante gespeichert wird, die nur für variante Tupel einen anderen Wert als *1* annehmen kann. Danach folgen die einzelnen Tupelfelder in der Reihenfolge ihrer Definition. Da TL-Tupel geordnete Bindungssequenzen darstellen [Matthes 92a], erlauben sie eine einstufige Repräsentation im Objektspeicher und indizierten Zugriff auf die TSP-Feldelemente.

### 4.3.4 Kontrollstrukturen

Die zwei TL-Kontrollstrukturen *if* und *case* werden durch eine elementare Operation '==' in TML dargestellt, die dem *switch*-Operator in C ähnelt. Die Operation '==' hat folgende Syntax:

(== val $val^t_1$ ... $val^t_n$ $val^c_1$ ... $val^c_n$ [$val^c_{n+1}$]) [3]

Der Wert *val* wird mit den Werten $val^t_1 \ldots val^t_n$ verglichen, und die entsprechende Fortsetzung aus $val^c_1 \ldots val^c_{n+1}$ ausgewählt. Die zusätzliche Fortsetzung $val^c_{n+1}$ stellt den *else*-Zweig in der Fallanalyse dar.

Das Konditional wird folgendermaßen nach TML übersetzt:

```
≪if A then     ≪A, ce, cont(a_1)                Auswertung der Bedingung
  B            (λ(join_2)
else             (== a_1 true false
  C                cont() ≪B , ce, join_2≫      then-Zweig
end                cont() ≪C , ce, join_2≫)     else-Zweig
D,             ≪D,ce,cc≫)≫                      Fortsetzung (join)
ce,
cc≫
```

Eine temporäre Variable (*join_2*) wird an die aktuelle Fortsetzung (mit der Berechnung von *D*) gebunden, um eine Codeduplizierung zu vermeiden.

Die Fallanalyse wird durch Aufruf der Operation '==' mit der Tupelvariante (einem Integer-Wert) implementiert:

```
≪case X                (λ(join_1)
  when mon, tue then   ≪X, ce, cont(x_2)
    A                  ([] x_2 0 cont(v_3)          Zugriff auf die Tupelvariante
  when wed, thu, fri then (λ(c_4 c_5 c_6)
    B                    (== v_3 1 2 3 4 5 c_4 c_4 c_5 c_5 c_5 c_6
  else                     cont() ≪A , ce, join_1≫
    C                      cont() ≪B , ce, join_1≫
end                        cont() ≪C, ce, join_1≫)
D,                     ≪D, ce, cc≫))≫)
ce,
cc≫
```

Da das Konditional ein Spezialfall der Fallanalyse ist, kann durch diese Unifizierung der beiden Kontrollstrukturen jede Verzweigung des Kontrollflusses im Code einheitlich behandelt werden.

[3] Mit $val^t$ werden die direkten Werte bezeichnet, mit $val^c$ die Fortsetzungswerte.

### 4.3.5 Ausnahmebehandlung

Ein großer Vorteil der CPS-Zwischensprache TML ist die einheitliche Behandlung der Ausnahmefortsetzungen und der normalen Fortsetzungen. Ein *raise*-Konstrukt wird einfach auf eine Applikation der aktuellen Ausnahmefortsetzung mit dem Ausnahmepaket abgebildet:

```
≪raise E with V end, ce,cc≫   ≪E, ce,  cont(e_1)
                               ≪V,  ce  cont(t_2)
                               (vector  e_1  t_2  cont(exc_3)    Ausnahmepaket
                               (ce  exc_3))≫≫                    raise
```

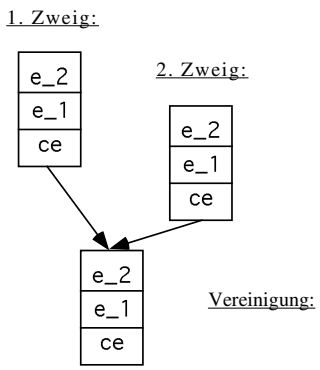
Hier ist das Ausnahmepaket durch *vector* gebildet, d.h. es ist unveränderlich. Daher könnte der Optimierer eventuell die Ausnahmevariable überprüfen, die aktuelle Ausnahmebehandlung ermitteln und Reduktion bzw. Expansion vornehmen.

Die Installation einer Prozedur zur Ausnahmebehandlung (*exception handler*) wird auf eine Bindung einer Abstraktion an den jeweiligen Fortsetzungsparameter abgebildet:

```
≪try                  (λ(join_1)
  f()                   (λ(e_2)
when E1 then H1          (f_99  e_2  join_1)
when E2 then H2         cont(exc_3)                  an e_2 gebundene Prozedur
else reraise             ≪E1 , ce,  cont(e_4)        zur Ausnahmebehandlung
end                      ≪E2,  ce,  cont(e_5)
D,                       ([]  exc  0  cont(excId_6)   Extrahieren des
ce,                                                   Ausnahmebezeichners
cc≫                      (==  excId_6  e_4  e_5
                           cont() ≪H1,  ce,  join_1≫
                           cont() ≪H2,  ce,  join_1≫
                           cont() (ce  exc_3)))≫≫)
                       ≪D, ce, cc≫ )
```

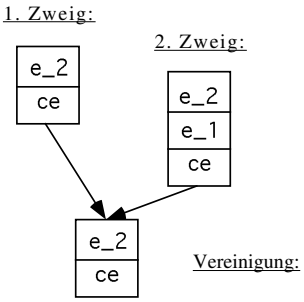
Bei der Ausnahme-Konvertierung wird die Ausnahmefortsetzung *e_2* aus dem Aufruf von *f_99* entfernt und statt dessen durch die elementare Operation *pushHandler* installiert. Der TML-Code sieht dann wie folgt aus:

```
proc (x_0 ce cc)
 ( lambda (e_1)
  ( lambda (f_3)
   ( lambda (e_2)
    ( lambda (c_4)
     (== x_0 true false
1. Zweig:
        cont () (f_3 e_1 cont (t_9)
          ... e_2 ... c_4)
2. Zweig:
        cont () ( ... e_1 ...
          ... e_2 ... c_4))
Vereinigung:
       cont () (... e_2 ... )

     cont (exc_5) ( ... )
  proc (e_6 c_7) ( ... ce ... c_6)
cont (exc_8) ( ... )
```




a) ohne Optimierung

```
proc (x_0 ce cc)
 ( lambda (e_1)
  ( lambda (f_3)
   ( lambda (e_2)
    ( lambda (c_4)
     (== x_0 true false
1. Zweig:
        cont () (... e_1 ... ce ...
          ... e_2 ... c_4)
2. Zweig:
        cont () ( ... e_1 ...
          ... e_2 ... c_4))
Vereinigung:
       cont () (... e_2 ... )

     cont (exc_5) ( ... )
  proc ( … ) ( … ce … )
cont (exc_8) ( ... )
```




b) mit Optimierung: Inlining von f_3 und Beta-Reduktion

Abbildung 4.3: Auswirkungen der Funktionsexpansion auf den Ausnahmestapel

```
≪try                    (λ(join_1)
  f()                     (λ(e_2)
when E1 then H1             (pushHandler  e_2  cont()      Installation einer Prozedur
when E2 then H2                                            zur Ausnahmebehandlung
else reraise                (f_99  cont(t_8)
end                         (popHandler  cont()            Synchronisation des
D,                                                         Ausnahmestapels
ce,                         (join_1  t_8))))
cc≫                       cont(exc_3)                      die Ausnahmebehandlung e_2
                          ≪E1 , ce,  cont(e_4)
                          ≪E2,  ce,  cont(e_5)
                          ([]  exc  0  cont(excId_6)
                          (==  excId_6  e_4  e_5
                            cont() ≪H1,  ce,  join_1≫
                            cont() ≪H2,  ce,  join_1≫
                            cont() (ce  exc_3)))≫≫)
                        ≪D, ce, cc≫ )
```

Es ist naheliegend, daß eine Optimierung die Verwaltung des Ausnahmestapels beeinflussen kann. Dieser Einfluß ist in Abb. 4.3 dargestellt, wo die Installierung von Ausnahmebehandlungen (*e_1*, *e_2* und *ce*) schematisch bei einer Codeverzweigung skizziert ist. Im unoptimierten Fall (Abb. 4.3.a) kann die Verwaltung recht einfach durch die Definitionsreihenfolge der Ausnahmevariablen bestimmt werden. Die Prozeduren zur Ausnahmebehandlung $ce$, $e_1$ und $e_2$ können nur in dieser Reihenfolge auf dem Laufzeitstapel liegen, da die Reihenfolge ihrer Verwendung genau der Reihenfolge ihrer Definition entspricht. Also ist die Abbildung auf Operationen auf dem Ausnahmestapel im ganzen Programm eindeutig durch die Definitionsreihenfolge der Prozeduren zur Ausnahmebehandlung vorgegeben, wobei der Stapel in seiner Tiefe unverändert bleibt.

In Abb. 4.3.b bringt die Expansion von *f_3* im ersten Zweig von '==' diese Ordnung der Prozeduren zur Ausnahmebehandlung auf dem Ausnahmestapel durcheinander. Obwohl die obersten Prozeduren in beiden Zweigen gleich sind (jeweils *e_2*), sind die Stapel am Vereinigungspunkt (Fortsetzung *c_4*) ungleich. Aus diesem Grund ist während der Ausnahme-Konvertierung eine Synchronisation der in den Zweigen gebildeten Ausnahmelaufzeitstapel an jedem Vereinigungspunkt des Programms notwendig.

### 4.3.6 Veränderliche Bindungen

TML-Variablen sind wie im Lambda-Kalkül nach ihrer Bindung unmodifizierbar. Veränderliche Bindungen müssen deshalb in explizite Manipulation des Objektspeichers über elementare TML-Operationen transformiert werden. Solche Bindungen werden folgendermaßen in TML abgebildet:

```
≪let var a = A, ce, cc≫   ≪A, ce, cont(t_1)        Berechnung von A
                          (array t_1 cont(a_2)     Initialisierung der Speicherzelle a_2
                          (cc t_1))≫               Ausgabe des Wertes
```

Während in TL Funktionen mit Referenzparametern möglich sind, besitzt TML als typische CPS-Repräsentation eine reine Wertübergabesemantik. Die Implementierung von **var**-Parametern (L-Werten) erfolgt durch ein Wertepaar aus der Basisadresse und dem Index der Speicherzelle:

```
≪let inc(var x :Int) =   (λ(inc_1)
  x := x + 1               (inc_1 a_2 0 ce cont(t_2)   Aufruf von inc mit
inc(a),                                               der Globalen a (a_2, s.o.)
ce,                        (cc t_2))
cc≫                      proc(x_3 xOffset_4 e_5 c_6)
                          ([] x_3 xOffset_4 cont(t_7)  Zugriff auf x
                          (+_50 t_7 1 cont(t_8)
                          ([]:=_91 x_3 xOffset_4 t_8   Modifikation von x
                                   e_5 c_6))))
```

Der Index (*offset*) kann ungleich Null werden, wenn Tupelkomponenten oder Feldelemente als L-Werte übergeben werden:

```
≪let k = array 1 2 3 end   (array 1 2 3 cont(k_1)
inc(k[2]),                 (inc_1 k_1 2 ce cont(t_2) (cc t_2)))   Index 2
ce,
cc≫
```

### 4.3.7 Bindung von Funktionen an elementare Operationen

Es existieren zwei Möglichkeiten, einfache TL-Operationen zu implementieren: durch Aufrufe externer C-Funktionen oder durch direkte Abbildung auf elementare TML-Operationen. Für den ersten Fall gibt es in TML die elementare Operation *ccall*, die eine externe C-Funktion aufruft (s. Abb. 4.2 auf Seite 33).

Für manche oft benutzte einfache Operationen wie z.B. Integer-Arithmetik ist es äußerst ineffizient, externe Implementationen (z.B. in C) zu benutzen. Aus diesem Grund bietet TL eine polymorphe Funktion *builtin* an, die der direkten Abbildung einer TL-Funktion auf eine elementare TML-Operation dient. Die *builtin*-Funktion hat folgende Signatur:

*builtin :***Fun**(**Dyn** *FctType <:Ok name :String ifFail :FctType) :FctType*

*ifFail* ermöglicht die Ausnahmebehandlung auf der TL-Ebene, so daß sie nicht im Übersetzer fest eingebaut ist. Der TL-Zuweisungsoperator ':=' ist z.B. folgendermaßen implementiert:

```
≪let {:=} =
 builtin(:Fun(A <:Ok var lhs :A rhs :A) :Ok
   ":="
   fun(A <:Ok var lhs :A rhs :A):Ok  B),
ce,
cc≫
```

```
(λ(:=_1)
 (cc  :=_1)
 proc(lhs_2  lhsOffset_3  rhs_4  e_5  c_6)
  (λ(fail_7)
   ([]:=  lhs_2  lhsOffset_3  rhs_4
     cont() (c_6  ok))
  cont()
   (λ(lhs_8  lhsoffset_9  rhs_10
              e_11  c_12)
    ≪B, e_11, c_12≫
   lhs_2
   lhsoffset_3
   rhs_4
   e_5
   c_6)))
```

Aus dem Beispiel ist zu erkennen, daß die Benutzung des TL-Zuweisungsoperators eine nicht ganz triviale TML-Funktion involviert, die auch Ausnahmebehandlung enthält. Da typkorrekte Zuweisungen nie fehlschlagen, kann die Berechnung $B$ als unerreichbaren Code entfernt und die Zuweisung auf den Aufruf der elementaren TML-Operation ':=' beschränkt werden.

Nachdem die Abbildung von höhersprachlichen Konstrukten auf die TML-Ebene vorgestellt wurde, kann auf die Maßnahmen zur Erzeugung eines effizienten TML-Code übergegangen werden. Die Optimierungstechniken und ihre Implementierung im Tycoon-System werden im nachfolgenden Kapitel diskutiert.

# 5 Optimierungstransformationen: Regeln und Realisierung

Dieses Kapitel stellt den Kern der vorliegenden Arbeit dar. In seinem ersten Teil werden die theoretischen Grundlagen der angewendeten Codeoptimierung mit Schwerpunkten auf Reduktion und Expansion (*Inlining*) diskutiert. Abschnitt 5.1 beschreibt die Grundregeln als Basis für die codeverbessernden Transformationen. In Abschnitt 5.2 werden Parallelen zu den in der Literatur bestehenden Optimierungstechniken gezogen, indem sie auf die in dieser Arbeit definierten Grundregeln abgebildet werden.

Im zweiten Teil des Kapitels wird konkret auf die Realisierung der Optimierungstransformationen im Rahmen des TL-Übersetzers eingegangen. Abschnitt 5.3 beschreibt die Reduktions- und Expansionsalgorithmen und präsentiert die verwendeten heuristischen Verfahren zur Codeanalyse. Im letzten Abschnitt (5.4) werden einerseits die Vereinfachungen verschiedener TL-Konstrukte durch die dynamische Optimierung dargestellt, und andererseits die Maßnahmen zur Steigerung der Effizienz des Optimierers selbst erläutert.

Nach der Überführung des TL-Syntaxbaumes in die TML-Repräsentation bedarf der Code einiger Transformationen, um effizienter gestaltet zu werden. Es existieren folgende drei Gründe, die eine Überarbeitung des direkt übersetzten TML-Codes sinnvoll machen:

1. Das TL-Bindungskonzept bedingt eine breite Verwendung von unveränderlichen Bindungen, die in der Programmierumgebung definiert sind und von Ausdrücken benutzt werden. Einfache Transformationen wie Substitution, Auswertung konstanter Ausdrücke, bzw. Entfernen unerreichbaren Codes können das TML-Programm bedeutend reduzieren und dadurch die Codeausführung beschleunigen. Außerdem sind selbst einfache Operationen wie '+', '−', '>', '<', etc. nicht implizit dem TL-Übersetzer bekannt, sondern werden über globale Bindungen zur Verfügung gestellt (s. Abschnitt 2.2). Ihre Expansion würde zu bedeutenden Codevereinfachungen führen.

2. Für jede benutzerdefinierte Funktion wird ein Funktionsobjekt (*closure*) gebildet und im Objektspeicher abgelegt. Der modulare Tycoon-Ansatz regt zu einem sauberen und aufgegliederten Programmierstil mit Verwendung vieler kleiner Funktionen an. Während dieser Ansatz einen entscheidenden Vorteil für das Systemdesign und die

Codewartbarkeit und -erweiterbarkeit bringt, wirkt er sich auf die Implementierung in einer großen Anzahl kleiner Funktionsobjekte aus, die zu Performanzeinbußen und zu erhöhtem Speicherplatzbedarf führen. Aus diesem Grund sind Transformationen wie Beta-Reduktion und Funktionsexpansion (*Inlining*) ein wesentlicher Faktor zur Kompensation der negativen Auswirkungen der Modularität auf die Implementation. Durch heuristische Verfahren wird der Benutzer von Optimierungsüberlegungen befreit. Dem Optimierer wird die Entscheidung überlassen, welche Funktion anstelle ihres Aufrufs substituiert werden soll.

3. Bedingt durch die implizite Persistenz des Tycoon-Systems kostet jeder Speicherzugriff erheblich mehr als in nichtpersistenten Systemen. Deswegen ist es wünschenswert, daß Objektspeicherzugriffe auf ein Minimum reduziert werden. Dieses Ziel ist z.B. durch Auswertung konstanter Ausdrücke und durch Eliminierung wiederholter Zugriffe auf dieselbe Variable (Eliminierung gemeinsamer Teilausdrücke) zu erreichen. Insbesondere trifft diese Überlegung für den Zugriff auf Bindungen aus importierten Moduln zu – eine Aktion, die für ein modulares System äußerst häufig wiederholt wird. Da ein Modul durch eine unveränderliche Tupelstruktur repräsentiert wird (s. Abschnitt 2.3), werden die exportierten Funktionen und Bindungen immer durch zwei Speicherzugriffe ermittelt: den ersten auf den Modul, und den zweiten auf die Bindung selbst. Durch Auswertung von '[]' kann der erste Zugriff eliminiert werden (vgl. Abschnitt 5.2.2).

Diese Aufgaben lassen sich mit der Reduktions- und Expansionsphase des TL-Optimierers lösen, der aus dem anfangs uneffizienten TML-Code eine kleinere und schneller ausführbare TML-Repräsentation erzeugt. Eine optimale bzw. beste Lösung ist für den resultierenden TML-Code unmöglich, da das Problem der Suche nach der effizientesten Repräsentation NP-vollständig ist. Der Begriff „Optimierung“ selbst ist in diesem Sinn irreführend und bezeichnet im Übersetzerbau Codetransformationen, die eine schneller laufende, bzw. weniger speicheraufwendige Programmrepräsentation erzeugen sollen.

Um den Optimierungsaufwand in Grenzen zu halten, konzentrieren sich die Bemühungen auf bestimmte Programmfragmente und Probleme, für die eine möglichst effiziente Form gesucht wird. Der Schwerpunkt des Optimierers leitet sich von solchen Eigenschaften des Tycoon-Systems wie Persistenz und Modularität sowie von seinem Bindungs- und strengen Typisierungskonzept ab und besteht vor allem in der Funktionsexpansion, in der Codereduktion und in der Minimierung der Speicherzugriffe. Die Bestrebung nach Portabilität, d.h. System- und Maschinenunabhängigkeit, von Tycoon hat auch die Grundlage für die Optimierung – die TML-Zwischenrepräsentation – prädestiniert.

Im Vergleich zu einer alternativen Quellsprachenoptimierung weist die ausdrucksvolle und einfache Zwischensprache TML bedeutende Vorteile auf. Auf der einen Seite werden verschiedene höhersprachliche Konstrukte wie Kontrollstrukturen, Schleifen und Datenstrukturen in eine einheitliche Darstellung auf der TML-Ebene abgebildet, so daß sie erkennbar bleiben. Damit lassen sich semantische Transformationen wie Tupel- bzw. Feldzugriffe sowie

Schleifen- und Feldtransformationen aufgrund konstanter Grenzwerte durchführen.

Außerdem profitiert der TL-Optimierer von der ihm vorausgehenden statischen Typüberprüfung, die eine Typanalyse zur Eliminierung von Laufzeittests während der Optimierung überflüssig macht.

Eine alternative Optimierungsumgebung wäre die Objekt- bzw. Maschinencoderepräsentation, deren Bedeutung besonders in [Davidson 86] hervorgehoben wird. Zwei Punkte sprechen für diese späte Optimierungsphase:

- Durch das Kennen der Maschinenressourcen können insbesondere Maschineninstruktionen, Adressierungsmodi und hardwarespezifische Werkzeuge (z.B. *pipelines*, *cache*, asynchrone Funktionseinheiten) effektiv ausgenutzt werden. Sehr gute Ergebnisse liefern in dieser Hinsicht die Registerallokation, die *peephole*-Optimierung und das *Targeting*.
- Nach der Codegenerierung kann ein neuer Bedarf an Optimierung entstehen.

In der Literatur existieren Beispiele, die das Wissen über die Maschinenarchitektur in die Zwischenrepräsentation einbeziehen. In EPIC [Kessler et al. 86] werden tabellengesteuerte maschinenabhängige Informationen in Form von RTL-Instruktionen verwendet. Orbit [Kranz et al. 86] nimmt sogar schrittweise Transformationen der CPS-Coderepräsentation vor, indem in der letzten Phase die Informationen über physikalische Maschinenregister zum Zweck der Registerallokation in den Code einfließen.

Das schon erwähnte Bestreben nach Portabilität im Tycoon-System gestattet jedoch dem Optimierer nicht, maschinenspezifische Informationen bei den Codetransformationen einzubeziehen und damit den TL-Übersetzer von irgendeiner Hardware-Plattform abhängig zu machen. Eine Kompensation dieser Nicht-Berücksichtigung der Maschinenressourcen bietet die zur interpretativen Ausführung alternative Möglichkeit der Übersetzung des TML-Codes nach C (s. [Mathiske 92]), wo die Optimierungseigenschaften des jeweiligen C-Übersetzers ausgenutzt werden können.

Es gibt eine Reihe von Anforderungen an die optimierenden Codetransformationen:

1. Die Transformationen müssen die Programmsemantik vollständig beibehalten, d.h. daß die Funktionalität eines Programms mit oder ohne Optimierung dieselbe bleiben soll. Dieses Kriterium besitzt in der Regel die höchste Priorität bei optimierenden Übersetzern. Man spricht vom *konservativen* Ansatz bei der Optimierung, der die Erhaltung der Funktionalität des veränderten Programms auf jeden Fall garantiert.
2. An zweiter Stelle steht die Anforderung, daß die Transformationen im Durchschnitt einen meßbaren Vorteil erbringen. Dabei wird neben der Verbesserung der Laufzeitgeschwindigkeit auch auf die Reduzierung des Speicheraufwands des generierten Codes Wert gelegt. Selbstverständlich existiert kein Optimierer, der für jedes Programm zu

einer deutlichen Verbesserung der Performanz führt. Die Güte eines Optimierers wird vielmehr an der durchschnittlichen Geschwindigkeitssteigerung (bzw. der Codegrößereduzierung) für eine repräsentative Auswahl von Programmen gemessen.

3. An dritter Stelle kommt die Überlegung nach dem Optimierungsaufwand. Die verbesserten Codeeigenschaften werden durch einen zusätzlichen Aufwand an Übersetzungszeit und Speicher erkauft, die in vertretbaren Grenzen gehalten werden müssen. Im allgemeinen könnte die Notwendigkeit einer Optimierung von der Häufigkeit, mit der ein Programm aufgerufen wird, und von seiner Bedeutung für den Benutzer abgeleitet werden. Offensichtlich sind oft gestellte Datenbankanfragen und generierte Funktionen (wie z.B. der TL-Parser – s. Kapitel 7) eher Kandidaten für eine Optimierung, als einmal oder selten verwendete Funktionen, da der Anteil des zusätzlichen Optimierungsaufwands bei den ersten, relativiert zum gesamten Zeitgewinn bei der Ausführung der optimierten Version, gegen Null neigt. Aus diesem Grund wäre der richtige Ansatz, die Entscheidung für eine zusätzliche Optimierung dem Ermessen des Benutzers selbst zu überlassen.

In der vorliegenden Arbeit sollen Lösungen für diese Anforderungen angeboten werden. Ohne einen formalen Korrektheitsbeweis für die optimierten Funktionen vorzulegen, wird das erste und wichtigste Kriterium dadurch erfüllt, daß die Korrektheit der Optimierungstransformationen durch die Grundregeln sichergestellt wird. Ihre Korrektheitszusicherungen werden von der CPS-basierten Sprache TML erleichtert. Außer bei der Expansion resultiert die Anwendung der Grundregeln in einen reduzierten Code, der auch schneller auszuführen ist. Dadurch wird sowohl das zweite Kriterium für die Effizienzsteigerung als auch die Terminierung des Optimierers sichergestellt. Die unkontrollierte Codeexpansion wird durch ein heuristisches Verfahren zur Kosten- und *Savings*-Abschätzung vermieden (s. Abschnitt 5.3.2.1).

Die Frage der Zweckmäßigkeit der Optimierung wird den Kompetenzen des Benutzers überlassen, indem er den dynamischen Optimierer über eine TL-Funktion – *reflect.optimize* – mit einem zu optimierenden Objekt (Funktion, Modul oder Tupel) aufrufen kann (s. Abschnitt 3.2).

In Kapitel 7 werden die Effekte des TL-Optimierers auf eine repräsentative Auswahl von Programmen gemessen und die Ergebnisse beurteilt.

## 5.1 Grundregeln des TL-Optimierers

Die codeverbessernden Transformationen, die der Optimierer ausführt, haben eine kleine Anzahl von Regeln als Grundlage, die auf bestimmte TML-Codefragmente anzuwenden sind und im folgenden in einer an [Plotkin 81], [Cardelli 90] und [Matthes 92a] angelehnten Notation aufgeführt sind. Die Kombinationen dieser Grundregeln erlauben die Realisierung einer breiten Palette der in der Literatur beschriebenen Optimierungstransformationen (s.

Abschnitt 5.2).

Die Wirkung jeder Regel drückt sich i.a. in der Transformation eines TML-Codefragments innerhalb einer bestimmten Umgebung ($\rho$) in einen äquivalenten TML-Ausdruck bzw. in eine veränderte Umgebung aus, wobei die Anwendung der Regel auch von Parametern abhängig sein kann:

$$Regel ::= in : In \xrightarrow[\rho : Env,\ parameter : Var]{Regel\text{-}Name} \triangleright\, out : Out \quad \Big(Bedingungen\Big)$$

$$In \subseteq TML$$

$$Out \subseteq (TML \times Env\ ) \bigcup\ Env$$

Die Umgebung enthält bestimmte Hilfsinformationen wie die Anzahl der Variablenreferenzen, Kosten- bzw. *Savings*-Informationen und vor allem Bindungen, die sich aus einer Lambda-Applikation bzw. PTML-nach-TML-Transformation ergeben.

Die Semantikerhaltung des gesamten TML-Programms basiert auf der Korrektheit der einzelnen Grundregeln. Die Einfachheit und die kleine Anzahl der Grundregeln erübrigen einen formalen Beweis, da es anhand der syntaktischen Definition von TML (s. Abb. 4.1 auf Seite 31) offensichtlich ist, daß sie korrekten TML-Code in einen korrekten TML-Code mit der gleichen Semantik transformieren, der sogar (außer bei der Expansion) immer kleiner als der Ausgangscode ist. Die Reihenfolge der Regelanwendung spielt für die Korrektheit keine Rolle, hat aber einen großen Einfluß auf die Effizienz des Optimierers. Diese wird in Abschnitt 5.4 behandelt. Im folgenden werden die einzelnen Regeln vorgestellt.

Die Regel *bind* hat die Aufgabe, Lambda-Variablen an die jeweiligen Argumente einer Lambda-Applikation bzw. eines $Y$-Konstrukts zu binden:

$$(\lambda\ (v_1 \ldots v_n)\ app\ val_1 \ldots val_n) \xrightarrow[\rho]{bind} \triangleright\, \rho,\ val_1/v_1 \ldots val_n/v_n$$

$$(Y\ \lambda\ (c\ t_1 \ldots t_n\ )(c\ \textbf{\textit{cont}}()app\ abs_1 \ldots abs_n\ )) \xrightarrow[\rho]{bind} \triangleright\, \rho,\ abs_1/v_1 \ldots abs_n/v_n$$

Diese Regel ist bedingungslos, ihr einziger Effekt ist eine veränderte Umgebung, die um die jeweiligen Variablen/Werte-Paare erweitert wird. Sie gilt auch im Fall einer Definition

wechselseitig rekursiver Funktionen mit der elementaren Operation $Y$ (s. Abschnitt 4.3.2), wo die Funktionsnamen an die jeweiligen Rümpfe gebunden werden.

Die eigentliche Substitution von TML-Variablen wird von der Regel *subst* realisiert:

$$v \xrightarrow[\rho,\ val/v]{subst} val, \rho' \quad \Big(val \notin Abs \bigvee |use(v)| = 1\Big)$$

Die Voraussetzung für diese Regel ist das Vorhandensein eines entsprechenden Variablen/ Werte-Paares in der Umgebung. Sie bezieht sich auf Variablen, die entweder nicht an Funktionen gebunden sind ($val \notin Abs$, d.h. Variablen- und Literalwerte werden ohne weiteres substituiert), oder auf Funktionsvariablen, die nur einmal im Code referenziert werden ($|use(v)| = 1$). Sie gehört zu den reduzierenden Transformationen, da selbst im zweiten Fall der Code nicht größer wird: der Rumpf der Funktion wird einfach von der Definitionsstelle an die Aufrufstelle verschoben. Diese Transformation wird in [Appel 92] Beta-Kontraktion (*beta contraction*) genannt. Eine Verletzung von Seiteneffekten ist bei der Anwendung von *subst* ausgeschlossen, da eine Lambda-Variable in TML nicht das Ziel einer Zuweisung (und damit kein Träger von Seiteneffekten) sein kann (s. Abschnitt 4.2.1).

Die Regel *inline* ist die einzige Regel, die zunächst zu einer Codevergrößerung führt. Die Schwierigkeit besteht jedoch nicht in der Anwendung der Regel selbst, sondern in der Schätzung der sich dadurch ergebenden Reduktionsmöglichkeiten des jeweiligen Funktionsaufrufs sowie in der Sammlung der dazu erforderlichen Informationen (s. Abschnitt 5.3.2).

*inline* hat zwei Ausprägungen:

- Variable an funktionaler Position, d.h. indirekter Aufruf eines Funktionsobjekts

$$(v\ val_1 \ldots val_n\ ) \xrightarrow[\rho,\ abs/v]{inline} (abs\ val_1 \ldots val_n\ ),\ \rho'$$

$$\Big(inlineCondition((v\ val_1 \ldots val_n\ ), \rho)\Big)$$

- *oid* an funktionaler Position, d.h. direkter Aufruf eines Funktionsobjekts

$$(oid\ val_1 \ldots val_n\ ) \xrightarrow[\rho,\ abs/oid]{inline} (abs\ val_1 \ldots val_n\ ),\ \rho'$$

$$\Big(abs = ptml2tml(oid) \bigwedge inlineCondition((oid\ val_1 \ldots val_n\ ), \rho)\Big)$$

Der erste Fall trifft für lokal im Codefragment definierte Funktionsbindungen zu. Der zweite Fall betrifft globale Funktionsaufrufe, tritt jedoch nur während der dynamischen Optimierung auf, da die Globalen nach dem Binden als Objektspeicherreferenzen (*oid*s) im TML-Baum dargestellt sind.

Die erste Voraussetzung für die Funktionsexpansion ist das Vorhandensein einer Variablen- ($abs/v$) bzw. *oid*-Bindung ($abs/oid$) an eine Abstraktion in der Umgebung. Die TML-Repräsentation (Lambda-Abstraktion) für eine Objektreferenz wird durch eine reflektive *ptml2tml*-Funktion aus dem Objektspeicher extrahiert, falls der dazu notwendige PTML-Code bei der vorausgegangenen separaten Übersetzung der Funktion generiert worden ist. Die zweite Voraussetzung für die codevergrößende Funktionsexpansion wird von einer Heuristik (*inlineCondition*) bestimmt, die von der Umgebung ($\rho$), von dem jeweiligen Funktionsaufruf und der zugehörigen Funktionsabstraktion abhängt (Abschnitt 5.3.2.1).

Die Regel *eliminateArgs* eliminiert in einer Lambda-Applikation die Argumente, deren jeweilige Lambda-Variablen im Rumpf der Abstraktion nicht verwendet werden:

$$(\lambda\ (v_1 \ldots v_n)\ app\ val_1 \ldots val_n) \xrightarrow[\rho]{eliminateArgs} (\lambda\ (v_1 \ldots v_{i-1} v_{i+1} \ldots v_n)\ app\ val_1 \ldots val_{i-1} val_{i+1} \ldots val_n),\ \rho' \quad \left(|app|_{v_i} = 0 \bigwedge 1 \leq i \leq n\right)$$

Das Prädikat $|app|_{v_i} = 0$ gibt die Bedingung für die Anwendung der Regel an, d.h. die Variable $v_i$ wird im Funktionsrumpf ($app$) nicht referenziert. Insbesondere kommt diese Regel bei der Eliminierung der für die TML-Syntax obligatorischen Ausnahmeparameter bzw. -argumente in Funktionen in Frage, die keine Ausnahmen auslösen.

Eine ähnliche Aufgabe wie *eliminateArgs* hat die Regel *Y-reduce*. Sie legt die sich nach erfolgten Optimierungen ergebenden von der normalen $Y$-Fortsetzung transitiv nicht referenzierten Funktionsdefinitionen ($trans(|app|_{v_i} = 0)$ in einem $Y$-Konstrukt fest und entfernt sie aus dem Code:

$$(Y\ \lambda\ (c\ v_1 \ldots v_n\ )(c\ \textbf{\textit{cont}}()app\ abs_1 \ldots abs_n\ ))\ \xrightarrow[\rho]{Y\text{-}reduce} (Y\ \lambda\ (c\ v_1 \ldots v_{i-1} v_{i+1} \ldots v_n)(c\ \textbf{\textit{cont}}()app\ abs_1 \ldots abs_{i-1} abs_{i+1} \ldots abs_n)),\ \rho' \quad \left(trans(|app|_{v_i} = 0)\ \bigwedge\ 1 \leq i \leq n\right)$$

Die einfachste und, allein angewendet, kaum einen Vorteil bringende Regel ist *removeAbstraction*:

$$(\lambda\ ()\ app\ ) \xrightarrow[\rho]{removeAbstraction} app,\ \rho'$$

Sie ersetzt eine Lambda-Applikation ohne Argumente durch den Funktionsrumpf. Die Bedeutung dieser Regel liegt in ihrer Anwendung als Bestandteil der Beta-Reduktion. Sie kann typischerweise nach einer wiederholten Eliminierung unbenutzter Funktionsparameter bzw. -argumente (durch *eliminateArgs*) die übrig gebliebene parameterlose Lambda-Abstraktion entfernen (s. Abschnitt 5.2.1).

In einer CPS-basierten Codedarstellung tritt oft der Fall auf, daß eine Fortsetzungsfunktion nichts weiter macht, als eine weitere Funktion mit denselben Argumenten aufzurufen. Bei einer solchen Übereinstimmung von Argumenten des Aufrufs und Parameter der Abstraktion wird die Regel *η-reduce* verwendet, um die überflüssige Lambda-Abstraktion zu entfernen:

$$\lambda\ (v_1 \ldots v_n\ )\ (v\ v_1 \ldots v_n\ ) \xrightarrow[\rho]{\eta\text{-}reduce} v,\ \rho'$$

Es kann durchaus vorkommen – besonders bei einer Sprache mit unveränderlichen Bindungen wie TL – daß die Argumente einer elementaren Operation nach einer Substitution zu Konstanten (Literalen in TML) auswerten. In so einem Fall kann man das Resultat der Anwendung einer solchen elementaren Operation noch während der Optimierung evaluieren und eventuell weiter verwenden bzw. eine aus mehreren Fortsetzungen (z.B. bei '==') auswählen. Die Regel *fold* bewirkt eine solche frühe Auswertung (*constant folding*) mit Hilfe der für jede elementare Operation spezifischen Meta-Evaluationsfunktion:

$$(p\ val_1 \ldots val_n\ ) \xrightarrow[\rho]{fold} eval\Big((p\ val_1 \ldots val_n\ )\Big),\ \rho'$$

In Abhängigkeit von den elementaren Operationen kann diese Funktion im Falle konstanter Argumente oder bestimmter Argumentbeziehungen zur Geltung kommen. Z.B. wird bei Anwendung von elementaren arithmetischen Operationen eine Lambda-Applikation aus der Fortsetzung und dem Ergebnis gebildet. Für die Operation '+' sieht die Auswertung konstanter Ausdrücke folgendermaßen aus:

$$(+\quad 1\ \ 2\quad \mathbf{cont}(t)\ app) \xrightarrow{fold} (\mathbf{cont}(t)\quad app\quad 3)$$

Von besonderem Nutzen ist die Auswertung konstanter Ausdrücke in zwei Situationen:

- Bei der Feldselektion auf ein unveränderliches Aggregat: Die Auswertung von '[]' zieht den Wert von der jeweiligen Feldposition heraus und erspart somit während der Laufzeit einen zusätzlichen Objektspeicherzugriff (vgl. Abschnitt 5.2.2).
- Bei der *if*- oder *case*-Kontrollstruktur, implementiert durch die elementare Operation '==' in TML: Bei einem Vergleich konstanter Argumente im Bedingungsteil können alle außer einem Fortsetzungszweig wegoptimiert werden. Dies entspricht dem Entfernen unerreichbaren Codes (*dead code removal*, vgl. Abschnitt 5.2.2).

In Verbindung mit der elementaren Operation '==' gibt es eine spezielle Regel (*case-assign*), welche die Reduktion in den einzelnen Zweigen erleichtert, indem sie die Bindung der Fallvariable an die Variantenwerte bestimmt:

$$(== \; v \; val_1^t \ldots val_n^t \; val_1^c \ldots val_m^c \;) \xrightarrow[\rho,\; i]{case\text{-}assign} \rhd \rho,\; val_i^t/v$$

$$\Big(1 \leq i \leq n \bigwedge n \leq m \leq n+1\Big)$$

$$(== \; v \; val_1^t \ldots val_n^t \; val_1^c \ldots val_m^c \;) \xrightarrow[\rho,\; n+1]{case\text{-}assign} \rhd \rho \quad \Big(n \leq m \leq n+1\Big)$$

Diese Regel verändert nur die Umgebung in der jeweiligen Verzweigung durch Hinzufügen eines Variable/Variante-Paares. Wenn auch ein *else*-Zweig angegeben ist, wird dort die Umgebung ($\rho$) unverändert weitergegeben, da in diesem Zweig der Wert der Fallvariable unbekannt ist. Im nächsten Abschnitt wird deutlich werden, daß die Regel *case-assign* die Grundlage für die *if*-Konvertierung schafft.

Die bisher eingeführten Transformationsregeln bilden das Gerüst der implementierten Reduktions- und Expansionsalgorithmen. Um einen Bezug zu den aus der Literatur bekannten Standardoptimierungen zu liefern, wird zunächst ihre Realisierung durch die oben angegebenen Grundregeln dargestellt.

## 5.2 Bezug zu Standardoptimierungstechniken

In der Literatur wird eine große Menge von Optimierungstechniken behandelt, die zu bestimmten Effizienzvorteilen führen können. Während einige davon mit wenig Aufwand durchzuführen sind, benötigen andere teuere Analyseverfahren, wie z.B. Seiteneffektanalyse bzw.

Datenflußanalyse. Im folgenden wird auf diese konventionellen Optimierungstransformationen eingegangen, die durch die im vorangegangenen Abschnitt beschriebenen Grundregeln realisiert werden. Die Mächtigkeit dieser recht kleinen Regelmenge reicht aus, um eine breite Palette von gängigen Optimierungen ohne den Einsatz einer Datenflußanalyse zu ermöglichen. Dabei wird von den Vorteilen der CPS-Repräsentation und ihrer Eigenschaften profitiert.

Die realisierten Optimierungstechniken können in fünf Gruppen klassifiziert werden, die im folgenden durch Beispiele in Bezug auf ihre Implementation im Tycoon-System diskutiert werden:

1. Optimierung von Funktionsaufrufen
2. Auswertung konstanter Ausdrücke
3. Konstanten-/Kopien-Propagierung
4. $\eta$-Reduktion
5. Vereinfachung rekursiver Funktionsdefinitionen

### 5.2.1 Optimierung von Funktionsaufrufen

Diese Gruppe von Optimierungstechniken nimmt einen zentralen Platz bei den Optimierungsbemühungen in Tycoon ein und betrifft i.a. das Ersetzen eines Funktionsaufrufs durch den Rumpf der aufgerufenen Funktion sowie auch die sich daraus ergebenden Codevereinfachungen. In der Literatur wird dieses Verfahren als (*Funktions-*)*Expansion* oder *Inlining* bezeichnet.

Eine solche Ersetzung des Funktionsaufrufs mit einer Abstraktion führt im ersten Moment zu einem Codewachstum, kann aber zu einem Laufzeitgewinn durch den *direkten* und den *indirekten Inlining*-Effekt [Dean, Chambers 93] führen: Der direkte *Inlining*-Effekt ist mit der Erübrigung der Zustandsspeicherung während des Kontextwechsels verbunden, der indirekte *Inlining*-Effekt umfaßt die Möglichkeiten zur weiteren Reduktion des substituierten Funktionsrumpfes nach der Parameterbindung. Es ist also eine Abschätzung dieser Effekte vorzunehmen und eine Entscheidung für oder gegen die Expansion zu treffen. Da es für den Optimierer problematisch ist, das genaue Laufzeitverhalten des zu optimierenden Programms vorauszusehen, bieten sich heuristische Verfahren an, die eine suboptimale, aber zufriedenstellende Lösung liefern. Ein solches Verfahren wird in Abschnitt 5.3.2.1 ausführlich beschrieben.

Die Grundregel *inline* implementiert direkt diese Transformation. Im Fall des Variablenaufrufs einer Funktion wird die Funktionsvariable mit der zugehörigen Abstraktion bei Erfüllung der *Inlining*-Bedingungen substituiert:

**TL:** **TML:**

```
let f(x :Int) = x          (λ(f_1)
f(1);                       (f_1 1 ce cc)
                           proc(x_2 e_3 c_4)
                            (c_4 x_2))
```

$\xrightarrow{inline}$

```
(λ(f_1)
 (λ(x_5 e_6 c_7)
  (c_7 x_5)
 1 ce cc)
proc(x_2 e_3 c_4)
 (c_4 x_2))
```

In diesem Fall erfolgt die Expansion des Aufrufs von *f* in zwei Schritten: zunächst wird durch die *bind*-Regel die Umgebung um die Bindung von *f_1* an die **proc**-Abstraktion erweitert, um im zweiten Schritt die erste Ausprägung der Regel *inline* anzuwenden (s. Abschnitt 5.1). Beim *Inlining* wird die Funktionsabstraktion an die Aufrufstelle kopiert, was an den umbenannten Parametern (*x_5* statt *x_2*, *e_6* statt *e_3* und *c_7* statt *c_4*) zu erkennen ist.

Im der zweiten Ausprägung der Regel *inline* wird die Erweiterung der Umgebung von der Funktion *ptml2tml* übernommen, die eine referenzierte Lambda-Abstraktion aus dem Objektspeicher (aus der PTML-Repräsentation) herausholt. Dieser Fall kann nur während der dynamischen Optimierung vorkommen, wenn das Binden schon durchgeführt ist und die Bezüge auf die globalen Referenzen hergestellt worden sind.

**TL:** **TML:**

```
2 + 3;    (<oid 0x006877dc>  2  3)
```

$\xrightarrow{inline}$

```
(λ(x_5 y_6 c_7)
 (+ x_5 y_6 cont(t_8) (c_7 t_8))
2
3)
```

$\rho$: *ptml2tml(<oid 0x006877dc>)* = **proc**(*x_1 y_2 c_3*) (+ *x_1 y_2* **cont**(*t_4*) (*c_3 t_4*))

Im obigen Beispiel wird die im Tycoon-System global definierte (Integer-)Additionsfunktion aufgerufen. Während der dynamischen Optimierungsphase stellt die reflektive Transformationsfunktion *ptml2tml* die TML-Abstraktion aus dem PTML-Code des mit dem *oid* referenzierten Funktionsobjekt wieder her (s. Abschnitt 3.2). Anschließend wird unter Anwendung der *Inlining*-Heuristik die Abstraktion anstelle der Speicherreferenz kopiert.

Die Expansion stellt einen aufwendigen Prozeß dar, weil der Funktionscode von der Stelle seiner Definition an die Stelle des Funktionsaufrufs kopiert wird. Dieses führt zur Coderepliкation. Im Fall einer einzigen Referenzierung einer Funktion kann diese teuere Transformation vermieden werden. Diesem Sonderfall der Expansion – der Beta-Kontraktion – wird in der Literatur spezielle Aufmerksamkeit geschenkt (s. [Appel 92]). Für seine Sonderbehandlung gibt es zwei Gründe:

- Da bei der Beta-Kontraktion eine Lambda-Abstraktion anstelle des Aufrufs verschoben (und nicht wie beim *Inlining* kopiert) wird, besteht keine Gefahr eines Codewachstums, so daß keine aufwendige *Inlining*-Entscheidung getroffen werden muß. Damit kann diese Transformation als Teil der Reduktionsphase aufgefaßt werden.
- In einer CPS-Repräsentation entsteht häufig ein solcher Bedarf. Zum einen wird er durch den von TL festgelegten Initialisierungszwang jeder Bindung (auch wenn sie einmal benutzt wird) bedingt, zum anderen durch Fortsetzungen, die einmal referenziert sind.

Die Beta-Kontraktion wird durch die aufeinanderfolgende Applikation der Grundregeln *bind* und *subst* in TML bewerkstelligt:

```
TL:                TML:

begin              (λ(f_1)                      subst ⟶   (λ(f_1)
  let f(x) = x       (f_1  1  ce  cc)                       (λ(x_2  e_3  c_4)
  f(1)             proc(x_2  e_3  c_4)                        (c_4  x_2)
end;                 (c_4  x_2))                            1  ce  cc)
                                                          (proc(x_2  e_3  c_4)
                                                            (c_4  x_2)))
```

Im Unterschied zum *Inlining*-Beispiel für die Identitätsfunktion, wird hier durch den vom **begin-end**-Rahmen bestimmten Sichtbarkeitsbereich die einmalige Verwendung der Funktion *f* garantiert. Der Code der Funktionsabstraktion wird anstelle der Variable *f_1* verschoben (worauf die gleichen Variablennamen in der Abstraktion an der Definitions- und Aufrufstelle zurückzuführen sind).

Nach dem *Inlining* bzw. nach der Beta-Kontraktion entsteht eine Lambda-Applikation, die durch eine *Beta-Reduktion* weiter vereinfacht werden kann:

```
(λ(x_1  y_2  e_3  c_4)
  (+  x_1  y_2  c_4)                  β- Reduktion ⟶   (+  a_6  b_7  cc)
a_6
b_7
ce
cc)
```

Diese Transformation besteht aus drei Schritten:

1. Substitution der formalen Parameter (*x_1*, *y_2*, *e_3*, *c_4*) durch die Argumente (*a_6*, *b_7*, *ce*, *cc*) im Funktionsrumpf. Sie wird durch die Regeln *bind* und *subst* realisiert und führt zu folgendem Zwischenergebnis:

```
(λ(x_1  y_2  e_3  c_4)
  (+  a_6  b_7  cc)
a_6
b_7
ce
cc)
```

2. Als Nächstes kann man die Regel *eliminateArgs* anwenden, um die im Funktionsrumpf unbenutzten Parameter und ihre entsprechenden Argumente zu entfernen:

   ```
   (λ()
     (+  a_6  b_7  cc))
   ```

   Diese Transformation führt einerseits zur Eliminierung unbenutzter Variablen (Parameter) und andererseits zum Entfernen unerreichbaren Codes (Wegwerfen von Lambda-Abstraktionen als Argumente). Außerdem werden durch die Regel *eliminateArgs* und die einheitliche Behandlung der Ausnahmefortsetzungen die unnötigen Ausnahmebehandlungen (*exception handlers*) aus dem Code ausgeschieden. In diesem Beispiel trifft das für den Handler *ce* zu, da der Parameter e_3 im Funktionsrumpf nicht referenziert wird.

3. Zuletzt kann man die Regel *removeAbstraction* anwenden und das Endergebnis ohne die überflüssige Lambda-Abstraktion bekommen, um weitere Vereinfachungen zu ermöglichen.

Eine Besonderheit der Sprache TML läßt eine weitere Optimierungstechnik auf das *Inlining* zurückführen – das Ausrollen von Schleifen. Wie es in Abschnitt 4.3.2 beschrieben ist, werden TL-Scheifen auf restrekursive Funktionen abgebildet, d.h. das Ausrollen einer Schleife ist nichts anderes als *Inlining* einer restrekursiven Funktion. Ein Beispiel folgt in Abschnitt 5.4.1.

### 5.2.2 Auswertung konstanter Ausdrücke

Es ist schon erläutert worden, daß die tatsächliche Auswertung konstanter Ausdrücke (*constant folding*) für eine elementare Operation von einer ihr zugeordneten Evaluationsfunktion realisiert wird. Die Regel *fold* gibt den allgemeinen Mechanismus des Einsatzes dieser Evaluationsfunktionen an. Sie können in verschiedenen Fällen aufgerufen werden:

- wenn die Wertargumente des Aufrufs der elementaren Operation Konstanten sind
- wenn bestimmte Abhängigkeiten zwischen den Wertargumenten auftreten
- wenn das Ergebnis im weiteren Programmverlauf nicht benutzt wird

- wenn mehrere Fortsetzungen dasselbe tun

In Abhängigkeit von der jeweiligen elementaren Operation und ihrer Evaluationsfunktion läßt sich eine Reihe interessanter Optimierungstechniken realisieren:

1. *if*- bzw. *case*-Vereinfachung. Bei diesen Transformationen handelt es sich am häufigsten um eine Entfernung der Operation '==' und die Auswahl einer ihrer Fortsetzungen, falls bestimmte Bedingungen erfüllt sind. Z.B. kann bei einem Konstanten-Vergleich folgende Situation vorkommen:

$$\begin{array}{l}(== \;\; 2 \;\; 1 \;\; 2 \;\; 3 \\ \quad \textbf{cont}() \;\; app_1 \\ \quad \textbf{cont}() \;\; app_2 \\ \quad \textbf{cont}() \;\; app_3)\end{array} \quad \xrightarrow{fold} \quad (\textbf{cont}() \;\; app_2)$$

   Mit anschließender Applikation von *removeAbstraction* bleibt dann nur $app_2$ aus dem ganzen Konstrukt bestehen. Es läßt sich erkennen, daß diese Transformation einen wichtigen Beitrag zum Entfernen unerreichbaren Codes leistet ($app_1$ und $app_3$ werden eliminiert).

   Es könnte sich herausstellen, daß nach der Reduktion der Fortsetzungsargumente gewisse Verzweigungen dieselbe Funktionalität aufweisen. Dann könnte die Fallanalyse aus dem Code entfernt werden:

$$\begin{array}{l}(== \;\; v\_1 \;\; t\_2 \;\; t\_3 \\ \quad \textbf{cont}() \;\; app \\ \quad \textbf{cont}() \;\; app)\end{array} \quad \xrightarrow{fold} \quad (\textbf{cont}() \;\; app) \quad \xrightarrow{removeAbstraction} \quad app$$

   Diese Transformation trägt mehr zur Reduktion des vom Code benutzten Speicherplatzes als zur Performanzsteigerung bei. Auf jeden Fall wird jedoch eine Vergleichsoperation zur Laufzeit eingespart.

2. Eine Anwedung der *fold*-Regel für die elementare Operation '==' in Verbindung mit einer vorausgegangenen Applikation der Regel *case-assign* führt zur von der Literatur bekannten *if*-Konvertierung:

$$\begin{array}{l}\textbf{if}\; p \;\textbf{then}\; f \\ \quad \textbf{elsif}\; p \;\textbf{then}\; g \\ \quad\quad \textbf{else}\; h \;\textbf{end}\end{array} \quad \xrightarrow{if\text{-}\,Konvertierung} \quad \textbf{if}\; p \;\textbf{then}\; f \;\textbf{else}\; h \;\textbf{end}$$

```
(== p_1 true false                  (== p_1 true false
  cont() (cc f_2)   --case- assign + subst-->   cont() (cc f_2)
  cont()                                        cont()
   (== p_1 true false                            (== false true false
    cont() (cc g_3)                                cont() (cc g_3)
    cont() (cc h_4)))                              cont() (cc h_4)))
```

```
--fold + removeAbstraction-->   (== p_1 true false
                                  cont() (cc f_2)
                                  cont() (cc h_4))
```

Wie es aus dem obigen Beispiel ersichtlich ist, werden zunächst die konkreten Ausprägungen der Fallvariable in den Zweigen der ersten elementaren Operation '==' substituiert (Regeln *case-assign* und *subst*). Dies ermöglicht die Anwendung der Regel *fold* für das zweite Auftreten von '=='. Nach *removeAbstraction* wird das Endergebnis geliefert. Nach diesem Verfahren werden eine Reihe von Regeln der booleschen Algebra realisiert, wie z.B. die Subsumption, Idempotenz oder die de-Morgan-Gesetze (s. Kapitel 6).

3. Wenn elementare arithmetische Operationen konstante Argumente erhalten, so wird das Ergebnis berechnet und mit der entsprechenden Fortsetzung (es könnte durchaus auch mit dem Ausnahmefall gerechnet werden, z.B. bei der Division durch *0*) eine Lambda-Applikation gebildet:

```
(+ 3 4 cont(t_1) app)   --fold'+'-->   (cont(t_1) app 7)
```

Anschließend kann durch Beta-Reduktion das konstante Argument in *app* weiterverwendet werden.

Auch wenn die Wertargumente einer elementaren Operation keine Konstanten sind, so kann durchaus passieren, daß die Ergebnisvariable der Operation im weiteren Programmlauf infolge vorausgegangener Reduktionen wegoptimiert wurde. Dann ist diese elementare Operation überflüssig:

$$(+\ v\_1\ v\_2\ \mathbf{cont}(t\_3)\ app) \xrightarrow{fold + eliminateArgs + removeAbstraction} app$$

$$(|app|_{t\_3} = 0)$$

4. Vergleichsoperationen können eventuell ausgewertet und die passende Fortsetzung ausgewählt werden:

$$\begin{array}{l}(>\ 5\ 4 \\ \quad \mathbf{cont}()\ app_1 \\ \quad \mathbf{cont}()\ app_2)\end{array} \xrightarrow{fold'>'} (\mathbf{cont}()\ app_1) \xrightarrow{removeAbstraction} app_1$$

   Im Unterschied zu den arithmetischen Operationen propagieren Vergleichsoperationen keinen Ergebniswert in den Verzweigungen, so daß die parameterlose Lambda-Applikation durch die Regel *removeAbstraction* eliminiert werden kann.

   Ein anderes Beispiel der Auswertung von Vergleichsoperationen ist die Feststellung bestimmter Abhängigkeiten zwischen den Argumenten, auch wenn diese keine Konstanten sind, z.B.:

$$\begin{array}{l}(>=\ x\_1\ x\_1 \\ \quad \mathbf{cont}()\ app_1 \\ \quad \mathbf{cont}()\ app_2)\end{array} \xrightarrow{fold'>='} (\mathbf{cont}()\ app_1) \xrightarrow{removeAbstraction} app_1$$

   Die Auswertungsfunktionen der elementaren TML-Operationen implementieren somit direkt einige triviale Eigenschaften der mathematischen Vergleiche (vgl. Abschnitt 6.3).

5. Die für eine persistente Umgebung wichtigste Anwendung der *fold*-Regel ist die Feldselektion von bekannten Datenstrukturen. Dadurch können einige der aufwendigen Objektspeicherzugriffe von der Laufzeit auf den Optimierungszeitpunkt vorgezogen werden. Eine solche Selektion ist nur dann möglich, wenn die elementare Operation '[]' als erstes Argument ein *oid*-Literal erhält, das die Referenz auf die Datenstruktur darstellt, und als zweites Argument einen konstanten Index (ein Integer-Literal). Außerdem soll das *oid* eine unveränderliche Struktur (TSP-Feld) referenzieren, damit durch den frühen Zugriff keine Seiteneffekte verletzt werden können. Durch den TSP-Felddeskriptor besteht in Tycoon die Möglichkeit solche Strukturen zu erkennen und somit noch zur Optimierungszeit Funktionen oder andere Objekte daraus zu selektie-

ren (s. Abschnitt 2.4). Folgendes Beispiel demonstriert den Zugriff auf Funktionen aus unveränderlichen Moduln:

***TL:***

```
bool.not(int.mul(2 3) > int.add(2 3));
```

***TML:***

```
([] < oid- bool > 8 cont(t_1)
([] < oid- int > 20 cont(t_2)
(t_2 2 3 cont(t_3)
([] < oid- int > 18 cont(t_4)
(t_4 2 3 cont(t_5)
(< oid-' >'> t_3 t_5 cont(t_6)
(t_1 t_6 cont(t_7)
(cc t_7))))))))
```

$\xrightarrow{fold\text{'[]'}+subst}$

```
(< oid- int.mul > 2 3 cont(t_3)
(< oid- int.add > 2 3 cont(t_5)
(< oid- ' >'> t_3 t_5 cont(t_6)
(< oid- bool.not > t_6 cont(t_7)
(cc t_7)))))
```

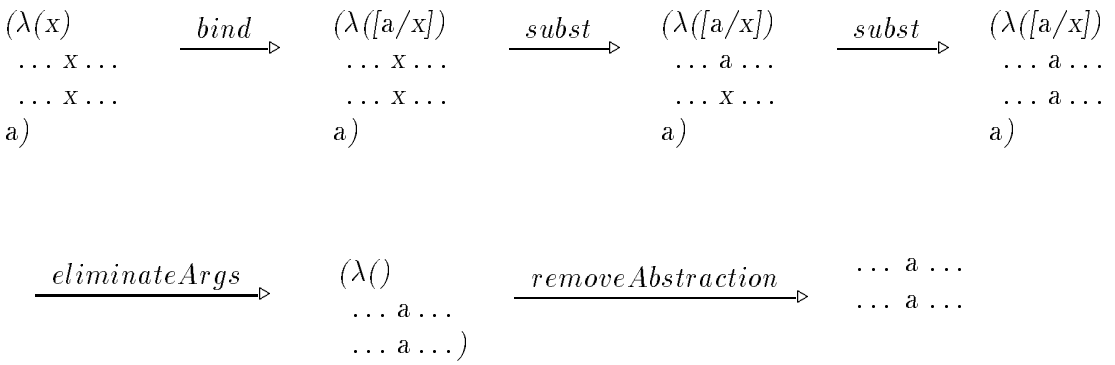
$\xrightarrow{inline + \beta\text{- Reduktion}}$ *(cc false)*

Dieses recht triviale Beispiel demonstriert einen bedeutenden Effekt der frühen Feldselektion, der über die Eliminierung eines teueren Speicherzugriffs hinausgeht. Diese Auswertung von '[]' ist vielmehr eine Voraussetzung für weitere Codevereinfachungen, besonders bei der dynamischen Optimierung, da dadurch der Optimierer die Objekte aus fremden Moduln bzw. anderen (unveränderlichen) Strukturen „sehen“ kann, und weitere Manipulationen wie *Inlining*, Beta-Reduktion oder Auswertung konstanter Ausdrücke vornehmen kann.

### 5.2.3 Konstanten-/Kopien-Propagierung

Das Bindungskonzept der Sprache TL unterstützt eher die breite Verwendung unveränderlicher Bindungen als die im imperativen Stil gängigen Zustandsvariablen. Die saubere Trennung in TML zwischen unveränderlichen Variablen (den Lambda-Variablen selbst) und den explizit durch TSP-Felddeskriptor als unveränderlich gekennzeichneten Objektspeicherstrukturen einerseits und den richtigen Zustandsvariablen andererseits gestattet die einfache

Analyse und Durchführung von Kopien-Propagierung (*copy propagation*) für die Bindungen. Da in imperativen Sprachen eher die Tendenz zu erkennen ist, daß Zustandsvariablen ausschließlich verwendet werden (auch für unveränderliche lokale Variablen, wofür z.B. in C die Deklaration **const** angeboten wird), kann man Kopien- bzw. Kostanten-Propagierung (*constant propagation*) nur nach einer aufwendigen Datenflußanalyse vornehmen. In TL erübrigt sie sich für eine große Zahl von Variablen, die durch die normalen (unveränderlichen) Bindungen erzeugt worden sind. Diese Transformationen werden durch die Regelpaare *bind* und *subst* (bzw. *case-assign* und *subst* für *if*- bzw. *case*-Verzweigungen) realisiert. Im Falle einer Konstantensubstitution ergibt sich die Konstanten-Propagierung und bei einer Variablensubstitution die Kopien-Propagierung. Hier ist ein Beispiel für die Kopien-Propagierung:




### 5.2.4 $\eta$-Reduktion

Wenn der einzige Zweck einer Abstraktion darin besteht, ihre Parameter als Argumente einer anderen Funktion zu übergeben, kann diese Abstraktion wegfallen. In der Literatur wird diese Transformation als $\eta$-Reduktion bezeichnet. Der TL-Optimierer realisiert sie durch die entsprechende Regel *$\eta$-reduce*. Wie schon in Abschnitt 5.1 erwähnt wurde, kommt ein solcher Fall in einer CPS-Zwischensprache beim Übergang von einer auf eine andere Fortsetzung häufig vor. Eine solche implizite Weitergabe von Funktionsargumenten tritt auf, wenn man neue Namen für Funktionen aus importierten Moduln einführt, um die Schreibarbeit und Lesbarkeit zu erleichtern. Ein typisches Beispiel dafür ist die Bindung der dynamischen Optimierungsfunktion:

***let*** *optimize* = *reflect.optimize*

### 5.2.5 Vereinfachung rekursiver Funktionsdefinitionen

Durch parallele Bindung in TL definierte, wechselseitig rekursive Funktionen werden in TML durch die elementare Operation *Y* implementiert. Im Laufe der Optimierungen kann

sich jedoch herausstellen, daß manche dieser rekursiven Funktionen nicht mehr referenziert werden. Dadurch verwandeln sie sich in unerreichbaren Code (*dead code*), und ihre Lambda-Abstraktion kann eliminiert werden. Die zweite Ausprägung der Regel *eliminateArgs* drückt diese Transformation aus (s. Abschnitt 5.1).

Falls es zu einer vollständigen Eliminierung der Funktionsbindungen im $Y$-Konstrukt kommt, kann $Y$ gestrichen werden:

$$(Y\ \textbf{\textit{proc}}(c)\ app) \xrightarrow{fold'Y'} app$$

Dies ist eine weitere Anwendung für die *fold*-Regel. In Abschnitt 5.4 wird ein Beispiel für die vollständige Evaluation von drei wechselseitig rekursiven Funktionen präsentiert.

## 5.3 Realisierung der Optimierungstransformationen

Der TL-Optimierer ist im *Back-End* des TL-Übersetzers integriert und manipuliert die TML-Zwischenrepräsentation mit dem Ziel einer effizienteren Ausführung zur Laufzeit. Derselbe Algorithmus wird sowohl in der statischen Phase während der separaten Übersetzung als auch in der dynamischen Phase nach dem Binden (s. Kapitel 3) eingesetzt.

Durch die Reflektion in Tycoon wird es möglich, daß alle Codefragmente aus verschiedenen Moduln in ihrer restaurierten TML-Zwischenrepräsentation dem Optimierer vorliegen. Aus diesem Grund kann bei der Betrachtung der Implementierung von dem jeweiligen Zeitpunkt der Optimierung (vor oder nach der Bindezeit) abstrahiert und der Algorithmus einheitlich vorstellt werden. Es wird jedoch auf einige wichtige Punkte hingewiesen, wo der dynamische Aspekt eine Rolle spielt.

Die Optimierung eines TML-Baumes läuft in zwei Phasen ab:

**Reduktionsphase:** In dieser Phase werden nur diejenigen Grundregeln angewendet, die nicht zu einer Vergrößerung des Codes führen. Dies sind alle Regeln bis auf *inline* (s.u.). Die Transformationen hier erzeugen also immer einen in Bezug auf die Laufzeit und auf den Speicheraufwand vereinfachten Code.

**Expansionsphase:** In dieser Phase wird die Regel *inline* angewendet. Sie führt trotz mancher Laufzeitvorteile (wegen der Entfernung des Mehraufwands eines Funktionsaufrufes) zunächst zu einem vergrößerten TML-Baum, da die Abstraktion anstelle des Aufrufs kopiert wird. Deswegen wird eine Analyse der Codegröße (Kosten) und der erwarteten Einsparungen (*Savings*) vorgenommen. Während der Traversierung des TML-Baumes werden die für die Regel notwendigen Informationen ermittelt, so daß die eigentliche Entscheidung keinen großen Aufwand darstellt.

Diese Phasen werden abwechselnd, beginnend mit der Reduktion, auf dem Code ausgeführt, bis sich in der Expansionsphase keine Codeänderungen mehr ergeben. Die Notwendigkeit dieser zyklischen Vorgehensweise folgt aus den sich nach jedem *Inlining* erschließenden Möglichkeiten, weitere Reduktionen auf dem substituierten Funktionsaufruf vorzunehmen. Um diesen Prozeß terminieren zu lassen, wird eine Heuristik verwendet (s. Abschnitt 5.3.2.1), welche die Wahrscheinlichkeit für ein erfolgreiches *Inlining* mit jedem weiteren Zyklus des Optimierers verringert.

Die beiden Phasen des Optimierungsalgorithmus sollen unter folgenden Gesichtspunkten betrachtet werden:

1. Bleibt die Semantik des Codes nach der Optimierung erhalten?
2. Mit welcher Strategie werden die Grundregeln aus Abschnitt 5.1. angewendet?
3. Ist die Terminierung des Algorithmus garantiert?
4. Wie effizient arbeitet der Optimierer selbst?

Die Antwort auf die erste Frage wird von der Korrektheit der einzelnen Grundregeln abgeleitet. Da jede davon einen korrekten TML-Baum in einen anderen korrekten TML-Baum mit der gleichen Funktionalität umwandelt, spielt die Reihenfolge der Regelanwendung für die Semantikerhaltung keine Rolle. Vielmehr ist sie von Bedeutung für die Effizienz des Optimierers selbst. Die Maßnahmen zu ihrer Steigerung werden in Abschnitt 5.4 diskutiert. Die Effizienz des Optimierers hängt vor allem von der Strategie der Regelanwendung ab, die für jede Phase separat (in Abschnitt 5.3.1 und Abschnitt 5.3.2.2) behandelt wird.

Die prinzipielle Terminierung ist für die erste Phase (Reduktion) immer garantiert, da der Code durch jede einzelne Regel immer kleiner als der Ursprungscode (oder höchstens gleich bei der Beta-Kontraktion) wird. Komplizierter sieht das Terminierungsproblem bei der Expansionsphase aus. Die darin verwendete heuristische Regel (s. Abschnitt 5.3.2.1) macht das *Inlining* von der Codegröße der zu substituierenden Lambda-Abstraktion und von den auf dem Code durchgeführten Optimiererzyklen abhängig und schließt dadurch eine unendliche Expansion (z.B. von rekursiven Funktionsaufrufen) aus.

Obwohl die Terminierung ein zentrales Problem der Algorithmentheorie darstellt, ist die Effizienz des Optimierers (die die Terminierung impliziert) von einer größeren praktischen Bedeutung. Der Benutzer erwartet, daß er eine Funktion bzw. einen Modul in absehbarer Zeit optimieren kann. Deswegen wird den Maßnahmen zur Effizienzsteigerung der Optimierung selbst extra ein Abschnitt (5.4) gewidmet, wo außer der speziellen Strategie der Regelanwendung in jeder Phase auch andere effizienzsteigernde Überlegungen beim Algorithmusdesign angesprochen werden.

### 5.3.1 Die Reduktionsphase

Im Anhang B.2 ist die vollständige formale Beschreibung des Algorithmus für die erste Phase der Optimierung – die Reduktionsphase – dargestellt. Die verwendete Notation stellt eine Erweiterung der bisher für die Grundregeln benutzen Notation dar und lehnt sich an [Plotkin 81], [Cardelli 90] und [Matthes 92a] an. Sie ist im Anhang B.1 zu finden. Im folgenden wird auf die wichtigsten Schritte dieses Algorithmus eingegangen.

Die Eingabe für die Reduktionsphase ist ein TML-Baum, genauer gesagt ein Lambda-Knoten (s. die TL-Schnittstelle für die TML-Datenstruktur im Anhang A). Jeder auszuwertende TL-Ausdruck wird in TML als eine Lambda-Abstraktion dargestellt, die als Parameter die aktuelle Tycoon-Umgebung (den *Top-Level*-Vektor) erhält.

Die Ausgabe nach der Reduktionsphase ist ein TML-Baum (wieder ein Lambda-Knoten), der sich nicht mehr reduzieren läßt. D.h. also, daß in einem Lauf die verschiedenen Reduktionsregeln angewendet werden, wobei manche davon in entgegengesetzten Richtungen ihren Einfluß haben. Besonders deutlich ist das der Fall bei der Beta-Reduktion (die Regeln *bind*, *subst*, *eliminateArgs* und *removeAbstraction*, in dieser Reihenfolge) und bei der Auswertung konstanter Ausdrücke (*fold*). Auf der einen Seite spricht die Beta-Reduktion für eine Bottom-Up-Traversierung des Baumes, da der Lambda-Applikationsknoten selbst erst dann reduziert werden kann, nachdem die Argumente, inclusive der Fortsetzungen (d.h. des ganzen Baumes darunter), schon reduziert worden sind. Die Reduktion der jeweiligen Lambda-Applikationen kann ihrerseits weitere Reduktionen nach oben verursachen.

Anders sieht es bei der Regel *fold* für die Auswertung konstanter Ausdrücke verschiedener elementarer Operationen aus. Sind die Argumente z.B. Konstanten, kann die elementare Operation frühzeitig evaluiert werden, wobei das Ergebnis weiter in die Fortsetzungen propagiert werden kann (Konstanten-Propagierung), um dort weitere Evaluationen zu initiieren. Die natürliche Richtung der Auswertung konstanter Ausdrücke ist folglich Top-Down. Leider können sich beide Reduktionen gegenseitig beeinflußen, was durch folgendes Beispiel deutlich wird:

```
proc (ce cc)
 (λ(f_1)
   (array f_1 cont(r_2)
   (f_1 4 5 cont(t_3) (cc t_3))
 proc(x_4 y_5 c_6) (+ x_4 y_5 cont(t_7) (c_6 t_7)))
```

Während der Reduktionsphase kann die Variable *f_1* zunächst nicht durch den Rumpf der daran gebundenen Funktion substituiert werden, da die Bedingung für die Regel *subst* nicht erfüllt ist (die Funktionsvariable wird zweimal verwendet). Erst in der Expansionsphase kann (durch *inline*) eine Expansion erfolgen, wobei auf die nächste Reduktionsphase gewartet werden muß, um die daraus resultierten Reduktionsmöglichkeiten zu realisieren.

Da aber die Variable *r_2*, die das durch *array* gebildete Feld referenziert, im Code nicht

weiter verwendet wird, kann die *array*-Applikation wegfallen. Damit bleibt *f_1* nur noch einmal referenziert und die Beta-Kontraktion kommt zur Geltung. Sie bringt die Möglichkeit zur Auswertung der Operation '+', so daß letztendlich nur folgendes übrigbleibt:

***proc*** *(ce cc)*
*(cc 9)*

Dieses Beispiel soll motivieren, daß alle Reduktionsmöglichkeiten ausgeschöpft werden sollten, bevor zu der deutlich teureren Expansionsphase übergegangen wird. Der Reduktionsalgorithmus ist so gestaltet, daß er nach jeder vollzogenen Reduktion die Traversierungsrichtung zu wechseln versucht (nach Auswertung konstanter Ausdrücke wird Beta-Reduktion versucht und umgekehrt).

Die zentrale Rolle bei der Implementierung der Reduktion spielt die Regel *reduce*, deren Aufgabe es ist, den Baum nach reduzierbaren Ausdrücken (*reducable expressions* oder auch *redexes*) zu untersuchen. Dementsprechend sind für diese Regel die entsprechenden Aktionen induktiv für alle Knotenarten definiert, wobei sich die Einfachheit und Minimalität der TML-Repräsentation in den wenigen Ausprägungen auswirkt.

Die elementaren Operationen und Literale lassen sich als Blattknoten ansehen, wo es keine weiteren Aktionen gibt. Jeder Lambda-Knoten wird zunächst in seinem Rumpf reduziert, d.h. *reduce* wird auf den Rumpf rekursiv angewendet. Anschließend wird die Grundregel *$\eta$-reduce* auf den schon modifizierten Lambda-Knoten benutzt.

Im Falle eines Variablenknotens wird die Variable durch den vorher in der Umgebung durch *bind* gebundenen Wert ersetzt. Falls dies wiederum eine Variable ist, wird die Regel *reduce* rekursiv auf die neue Variable aufgerufen (s. Regel *substRec*). Dieser Prozeß stellt die Kopien-Propagierung von Bindungen dar.

Der letzte TML-Knotentyp ist der Applikationsknoten, der vier verschiedene Ausprägungen haben kann: Literal-, Variablen-, Lambda-Applikation bzw. Aufruf einer elementaren Operation. Im ersten Fall, Literal-Applikation, wird direkt zur Reduktion der Argumente übergegangen. Beim Variablen-Aufruf einer Funktion wird versucht, die Funktionsvariable zu substituieren, was letztendlich zu einer Lambda-Applikation führen kann. Dieser Fall (Beta-Kontraktion) kommt in Frage, wenn die Funktion ein einziges Mal im Code referenziert wird. Bei erfolgter Beta-Kontraktion wird der Lambda-Applikationsknoten weiter reduziert, sonst werden die Argumente traversiert.

Bei den Aufrufen elementarer Operationen muß die *Y*-Operation von den anderen unterschieden werden. Der Grund dafür ist, daß (anders als bei allen anderen elementaren Operationen, s.o.) die Anwendung der *fold*-Regel auf *Y* erst nach der Reduktion der Argumente (der rekursiven Lambda-Abstraktionen) sinnvoll ist. Demzufolge ist hier ein Bottom-Up-Verhalten zu beobachten. Also werden zunächst die Rümpfe der durch die Operation *Y* definierten, wechselseitig rekursiven Funktionen und dann die normale Fortsetzung reduziert. Darauf folgt die Anwendung der Regel *Y-reduce*, mit der versucht wird, Definitionen

von unbenutzten Funktionen zu eliminieren. Erst danach wird die Regel *tryFold* angewendet, welche die Auswertung konstanter Ausdrücke einleitet und eventuell (wenn dieses erforderlich ist) für die Umkehrung der Traversierungsrichtung durch die Beta-Reduktion sorgt.

Bei allen anderen elementaren Operationen wertet die *reduce*-Regel zunächst die Wertargumente aus und geht in die Regel *tryFold* über. Die schon angesprochene Regel *tryFold* führt als erstes die Auswertung konstanter Ausdrücke durch *fold* aus und ruft in Abhängigkeit von dem modifizierten TML-Knoten erneut *reduce* auf (zur Beta-Reduktion im Falle eines nach Auswertung der elementaren Operation entstandenen Lambda-Applikationsknotens) oder reduziert die nicht traversierten Fortsetzungen des unveränderten Aufrufs der elementaren Operation weiter. Wenn es sich im zweiten Fall um die Operation '==' handelt, werden die Fortsetzungen mit den von der Grundregel *case-assign* definierten Umgebungen in den Verzweigungen reduziert.

Die Beta-Reduktion wird von der auf eine Lambda-Applikation angewendeten Regel *reduce* verkörpert. Um die TML-Baumtraversierung auf ein Minimum zu bringen, wird hier eine Methode verwendet, die eher darauf ausgerichtet ist, die Lambda-Abstraktion an der Funktionsposition auszuwerten. Wenn das für die Beta-Reduktion spezifische, reine Bottom-Up-Verfahren angewendet wird, sollte die Reduktion der Lambda-Abstraktion erst nach der Reduktion aller Argumente, inclusive der Fortsetzungen (d.h. des ganzen unter dem zu bearbeitenden Knoten liegenden Baumes), erfolgen. Nach der Beta-Reduktion und einer eventuellen Auswertung des Aufrufs sollte man dann wieder in die entsprechende Fortsetzung übergehen, um z.B. die Propagierung des Ergebnisses weiterzugeben.

Das folgende Beispiel kann als Illustration dieses Ansatzes dienen, wo die Fortsetzungen des *f_2*-Applikationsknotens zweimal traversiert werden (vor und nach der Beta-Reduktion):

```
(λ(a_1  f_2)                          1. bind
 (f_2  a_1  2                         3. subst a_1;  6. subst f_2
   cont()  app                        4. reduce
   cont(t_3)                          5. reduce
    (+  t_3  8  cont(t_4)
    (cc  t_4)))
10
proc(x_5  y_6  e_7  c_8)              2. reduce
 (/  x_5  y_6
   cont()
    (raise  "division by zero")
   cont(t_9)
    (c_8  t_9)))
```

```
(λ(a_1  f_2)
 (λ(x_5  y_6  e_7  c_8)                    7. bind
  (/  x_5  y_6                             8. reduce (u.a. fold'/')  ⟶
   cont() (raise  "division by zero"))
   c_8
 10
 2
 cont()  app
 cont(t_3)
  (+  t_3  8  cc))
10
proc(x_5  y_6  e_7  c_8) ... )
```

```
(λ(a_1  f_2)
 (λ(x_5  y_6  e_7  c_8)          14. eliminateArgs;  15. removeAbstraction
  (cont(t_3)                     12. eliminateArgs;  13. removeAbstraction
   (+  t_3  8  cc)               10. subst t_3;  11. fold'+'  (2. Traversierung)
  5)                             9. bind t_3
 10
 2
 cont()  app
 cont(t_3)
  (+  t_3  8  cc))
10
proc(x_5  y_6  e_7  c_8) ... )
```

⟶ (*cc* 13)

Die Codetransformationen sind in 15 Schritten zusammengefaßt, um die Baumtraversierungen zu verdeutlichen. Es ist zu beachten, daß im 4. und 5. Schritt beide Fortsetzungen der Applikation von *f_2* ohne großen Erfolg traversiert werden (d.h., daß keine Reduktionen vorgenommen werden können). Erst nach der Evaluation des substituierten Funktionsrumpfes und nach der Auswertung von '/' kann der richtige Zweig für die weitere Bearbeitung ausgewählt und zum zweiten Mal erfolgreich reduziert werden.

Es ist klar, daß der Aufruf von *f_2* nach der Beta-Kontraktion (Regel *subst*) evaluiert werden

kann: nach der Substitution der Variable a_1 durch *10* liefert der Aufruf von / ein konstantes Ergebnis (*5*). Daraus ergibt sich logisch die Bemühung, so früh wie möglich die Reduktion der Funktionsabstraktion an der Aufrufstelle vorzunehmen.

Aus diesem Grund werden bei der Beta-Reduktion (Regel *reduce* auf einem Lambda-Applikationsknoten) zunächst nur die Lambda-Argumente über $\eta$-*reduce* vereinfacht, um unnötige Abstraktionen besonders in den Fortsetzungen zu eliminieren, und nach einer *bind*-Applikation der Rumpf der Abstraktion reduziert. Anschließend werden die unbenutzten Argumente entfernt. Die nicht eliminierten Argumente, die jetzt verbleiben, sind die nicht traversierten Abstraktionen der Fortsetzungen bzw. der Benutzerfunktionen, die mehr als einmal im Rumpf referenziert sind. Sie werden erst jetzt reduziert. Falls die Abstraktionsargumente zu Variablen oder Konstanten evaluieren (was sehr selten vorkommt, da die $\eta$-Reduktion schon stattgefunden hat), sollte die Beta-Reduktion erneut auf den Lambda-Applikationsknoten erfolgen.

Typischer ist der Fall, bei dem nach der Reduktion der Lambda-Abstraktion und der Eliminierung der unbenutzten Argumente gleich die Regel *removeAbstraction* anwendbar ist. Dadurch wird im obigen Beispiel, nach der Evaluation von der Funktion *f_2* ihre Ausnahmefortsetzung (mit *app* als Rumpf) von der Regel *eliminateArgs* als unerreichbaren Code noch vor ihrer Traversierung entfernt. Außerdem wird die verbleibende Fortsetzung insgesamt einmal und erfolgreich traversiert.

Durch die beschriebene Methode wird die unnötige, mehrmals wiederholte Bearbeitung des TML-Baumes vermieden, so daß die Zahl der traversierten Knoten nicht viel größer als die Kardinalität des TML-Baumes ist. Überraschenderweise kann diese Zahl sogar kleiner als die Baumkardinalität werden. Diese Eigenschaft des Algorithmus ist im letzten Beispiel zu erkennen. Der Vorteil ist um so größer, je größer der nicht traversierte, eliminierte Teilbaum ist (die Fortsetzung mit dem Rumpf *app*). Diese Überlegung ist der Kernpunkt des Effizienzgewinns des Optimierungsalgorithmus in der Reduktionsphase (s. Abschnitt 5.4).

### 5.3.2 Die Expansionsphase

Die Reduktionsphase liefert einen vollständig reduzierten TML-Baum, in dem jedoch die mehrmals referenzierten Funktionsaufrufe nicht substituiert sind. Diese Aufgabe übernimmt die sich daran anschließende Expansionsphase, die ausschließlich auf der Applikation der Grundregel *inline* beruht. Diese konzeptionelle Trennung ergibt sich aus den deutlich aufwendigeren Transformationen, die während der Expansion durchgeführt werden. Zum einen werden die zu ersetzenden Lambda-Abstraktionen an die Aufrufstelle kopiert, anders als die billige Verschiebung während der Beta-Reduktion. Zum anderen wächst dadurch der TML-Baum, wobei die Terminierung nicht wie bei der Reduktion garantiert ist.

Aus diesem Grund wird die Regel *inline* unter einer Bedingung (*inlineCondition*) eingesetzt, die von dem Aufruf, der Funktionsabstraktion und der Umgebung abhängig ist. Im Tycoon-System funktioniert die Expansion automatisch, d.h. die Entscheidung zur Appli-

kation der Regel *inline* wird vom Optimierer getroffen. Die andere Alternative ist, diese Entscheidung dem Programmierer zu überlassen (wie das der Fall beim MIPS-C-Übersetzer oder bei Ada-Übersetzern ist). Der Nachteil dieser Vorgehensweise besteht einerseits in der Belastung des Programmierers mit systemspezifischen Fragen, und andererseits in der ungenügenden Nutzung der Informationen des dynamischen Optimierers über die im Code verwendeten Funktionen, die weit über das Wissen des Programmierers (besonders bei komplexeren Systemen) hinausgeht. Daher werden in modernen optimierenden Übersetzern die Kompetenzen für das *Inlining* dem System überlassen ([Hwu, Chang 89], [Chang et al. 92], [Davidson, Holler 88], [Cooper et al. 91], [Chambers 92], [Appel 92]).

Im TL-Optimierer wird auch diese Methode gewählt, wobei der Optimierer automatisch, mit Hilfe einer Heuristik, die Entscheidung für oder gegen das *Inlining* trifft. Bevor der Expansionsalgorithmus vorgestellt wird, soll das Prädikat *inlineCondition* der Regel *inline* erläutert werden.

#### 5.3.2.1 Die Bedingung für das Inlining

Das *Inlining* hat zwei entgegengesetzte Auswirkungen auf den Code. Zum einen wird durch die Vermeidung des Mehraufwands bei der Aktivierung einer Funktion die Laufzeitperformanz verbessert. Dieser Effekt wird in [Dean, Chambers 93] als direkter *Inlining*-Effekt bezeichnet, wobei die Speicherung der momentanen Laufzeitumgebung auf dem Laufzeitstack vor dem Funktionsaufruf und ihre Reaktivierung nach dem Aufruf eingespart wird. Außerdem spielt das *Inlining* von Prozeduren eine entscheidende Rolle für die sich daraus ergebenden Möglichkeiten für weitere Codevereinfachungen. Wie in [Dean, Chambers 93] hervorgehoben wird, kann dieser indirekte Effekt eine viel bedeutendere positive Auswirkung auf das Laufzeitverhalten haben als der direkte *Inlining*-Effekt.

Zum anderen existiert die Gefahr, daß die Übersetzung (wegen der übereifrigen Optimierung) nicht terminiert, insbesondere bei rekursiven Funktionen, deren Aufrufe unendlich expandiert werden. Dies kann zu einem erhöhten Speicherbedarf und zu Seiteneffekten zur Laufzeit (z.B. *Swapping*) führen, welche die Vorteile der Expansion negativ kompensieren ([Davidson, Holler 88], [Cooper et al. 91], [Hwu, Chang 89], [Chang et al. 92]).

Folglich sollten zwei Kriterien im Vergleich zueinander untersucht werden: das mögliche effizientere Laufzeitverhalten und das Codewachstum.

Ähnlich wie bei [Appel 92] wird in dieser Arbeit der positive Effekt (Laufzeitgewinn) durch die *Savings* (Einsparungen), und der negative Effekt (Codewachstum) durch die *Kosten* repräsentiert. Für beide Kriterien werden als Maß abstrakte Maschineninstruktionseinheiten verwendet, die dem jeweiligen Zweck (Speicher- bzw. Laufzeitabschätzung) dienen und einen Vergleich zueinander erlauben. Die Entscheidungsregel für das *Inlining* ist in Abb. 5.1 angegeben.

Im allgemeinen werden die für das *Inlining* nachteiligen Kosten des Rumpfes gegenüber dem direkten *Inlining*-Effekt (dem durch das *Inlining* verschwindenden Mehraufwand des

**Aufruf:** $(val_0\ val_1 \ldots val_n\ )$
**Umgebung des Aufrufs:** $\rho,\ \lambda(v_1 \ldots v_n\ )app\ /\ val_0$

$$cost(app) - costOfCall\Big((val_0\ val_1 \ldots val_n\ )\Big) - \sum_{i=1}^{n} savings_{v_i}(app) - \frac{cost(app)}{|use(val_0)|}$$

$$<$$

$$optimisticFactor - nRounds * pessimisticFactor$$

Abbildung 5.1: Entscheidungsregel für das *Inlining*

Aufrufs) und den indirekten *Inlining*-Effekten (in den restlichen Termen auf der linken Seite der Ungleichung) abgewogen. Durch die *Savings* werden die sich nach dem *Inlining* und der Parametersubstitution ergebenden künftigen Reduktionen des Aufrufs abgeschätzt. Der letzte Term auf der linken Seite der Ungleichung erfaßt die Wahrscheinlichkeit für die künftige Eliminierung der Funktionsdefinition durch die Regel *eliminateArgs* mit jedem weiteren *Inlining*.

Je kleiner die linke Seite der Ungleichung wird, desto günstiger ist es, an dieser Stelle eine Ersetzung des Funktionsaufrufs durch die Lambda-Abstraktion vorzunehmen. Die rechte Seite gibt die Möglichkeit zur Steuerung des Terminierungsprozesses durch Optimierungsparameter an: durch den optimistischen Faktor, der sich für rekursive und nicht rekursive Funktionen unterscheidet, und durch den pessimistischen Faktor, dessen negativer Einfluß auf die Expansion sich mit jedem weiteren Optimiererzyklus verstärkt. In Kapitel 7 werden die Auswirkungen dieser Faktoren auf bestimmte Standardtestprogramme gemessen.

In Abb. 5.2 ist die Bestimmung der Kosten für die verschiedenen TML-Knotentypen dargestellt. Die Kosten eines Knotens setzen sich rekursiv aus den Kosten der darunterhängenden Teilbäume zusammen.

Sie werden in elementaren Operationen einer abstrakten Maschine gemessen, wo z.B. die Referenzierung einer Variable oder Konstante als eine Operation aufgefaßt ist, während die Ausführung einer Addition drei Operationen (zwei Allokationen für die Argumente und eine für das Resultat) kostet.

Während sich die Kosten des Rumpfes der einzusetzenden Funktion negativ auf die *Inlining*-Entscheidung auswirken, bringt die Substitution des Funktionsaufrufes den Vorteil, daß der Mehraufwand bei der Aktivierung der neuen und Reaktivierung der alten Funktion vermieden wird. Dieser direkte Effekt des *Inlining* ist in der Entscheidungsregel mit dem Term *costOfCall* bezeichnet und wirkt gegen den negativen Einfluß der Codegröße der

$$cost(v) = 1$$

$$cost(l) = 1$$

$$cost(p) = nResults(p) + nValueArgs(p)$$

$$cost\Big(\lambda(v_1 \ldots v_n\ )app\Big) = cost(app) + n$$

$$cost\Big((val_0\ val_1 \ldots val_n\ )\Big) = cost(val_0) + \sum_{i=1}^{n} cost(val_i)$$

Abbildung 5.2: Berechnung der Kosten des TML-Baumes

Funktionsabstraktion. Die Aufrufkosten werden folgendermaßen bestimmt:

$$costOfCall\Big((val_0\ val_1^t \ldots val_n^t\ val_1^c \ldots val_m^c\ )\Big) = cost(f) + n + m'$$

$$val_1^c \ldots val_{m'}^c \quad \subseteq \quad val_1^c \ldots val_m^c$$

$$\forall val_i^c \in \ val_1^c \ldots val_{m'}^c \ : val_i^c \in Var$$

Die Aufrufkosten werden von der Anzahl der Wertargumente und von der Anzahl der Variablen als Fortsetzungsargumente (die Sprünge darstellen), nicht aber von den Fortsetzungen selbst in Abhängigkeit gesetzt. Damit stellen sie die Erübrigung der für den Aufruf notwendigen Registerallokationen (den direkten *Inlining*-Effekt) dar.

Es ist leicht zu erkennen, daß der direkte *Inlining*-Effekt im Verhältnis zu den rekursiv bestimmten Kosten des Funktionsrumpfes ziemlich gering ausfällt. Wenn man nur diese Auswirkung des *Inlining* betrachtet, würden sich nur sehr wenige, extrem kleine Funktionen für eine Substitution eignen. Von einer viel größeren Bedeutung für das *Inlining* ist die daraus resultierende Gelegenheit zur weiteren Reduktion des Funktionsaufrufs sein. Diese Konsequenzen können jedoch nur durch eine Heuristik geschätzt werden, da die Entscheidung im voraus getroffen werden soll. Eine genaue Berechnung würde einen aufwendigen „Testlauf“ des Algorithmus erfordern. Ähnlich wie in [Appel 92] werden dazu die Einsparungen, *Savings*, als Maß für die möglichen Reduktionen in der Lambda-Abstraktion berechnet. Durch die Bindung der formalen Parameter an konstante Argumente (Literale) kann sich einerseits eine frühe Evaluation an einem Aufruf einer elementaren Operation im Funktionsrumpf ergeben. Andererseits kann eine Funktion höherer Ordnung (**proc**-Abstraktion als Argument) im Rumpf substituiert und ausgewertet werden.

$$savings_v(v') = 0$$

$$savings_v(l) = 0$$

$$savings_v\Big(\lambda(v_1 \ldots v_n\ )app\Big) = savings_v(app)$$

$$savings_v\Big((val_0\ val_1 \ldots val_n\ )\Big) = \sum_{i=0}^{n} savings_v(val_i)$$

$$savings_v\Big((p\ val_1 \ldots val_n\ )\Big) = \sum_{i=1}^{n} savings_v(val_i) \qquad v \notin val_1 \ldots val_n$$

$$savings_v\Big((p\ val_1^t \ldots val_n^t\ val_1^c \ldots val_m^c\ )\Big) = \qquad v \in val_1^t \ldots val_n^t$$

$$\sum_{i=1}^{n} savings_v(val_i^t)\ +\ \frac{1}{m} * \sum_{i=1}^{m} savings_v(val_i^c)\ +$$

$$\frac{|val_1^t \ldots val_n^t\ |_v}{\sum_{t \in Temp} |val_1^t \ldots val_n^t\ |_t} * (cost(p) + n - 1 + \frac{m-1}{m} * \sum_{i=1}^{m} cost(val_i^c))$$

Abbildung 5.3: Berechnung der *Savings* des TML-Baumes

In der Expansionsphase des TL-Optimierers werden die potentiellen *Savings* für jede Lambda-Variable während der Bottom-Up-Traversierung des TML-Baumes bestimmt. Ähnlich wie die Kosten werden die potentiellen Variablen-*Savings* aufgrund der abstrakten TML-Syntax induktiv definiert und akkumulieren den Beitrag jeder Lambda-Variable für eventuelle Codereduktionen, falls diese an eine Konstante gebunden wird (s. Abb. 5.3).

Blattknoten (Literale und Variablen) haben keinen Einfluß auf die Berechnung der *Savings*. Abstraktions- und Applikationsknoten dienen nur zur Propagierung der *Savings* durch den TML-Baum. Es ist naheliegend, daß es zu potentiellen Variablen-*Savings* nur an den Aufrufknoten elementarer Operationen kommen kann, wo eine Auswertung konstanter Ausdrücke möglich ist (s. Abschnitt 5.2.2). Deswegen wird im folgenden näher auf den Fall des Aufrufs einer elementaren Operation eingegangen (s. beide letzten Formeln in Abb. 5.3).

Wenn die untersuchte Variable $v$ nicht in der Liste der Wertargumente der elementaren Operation vorkommt, wird die Summe der in den Argumenten enthaltenen *Savings* für $v$ gebildet (vorletzte Formel für die *Savings*-Berechnung).

Falls aber $v$ ein Argument der elementaren Operation ist, werden zunächst nur die *Savings*

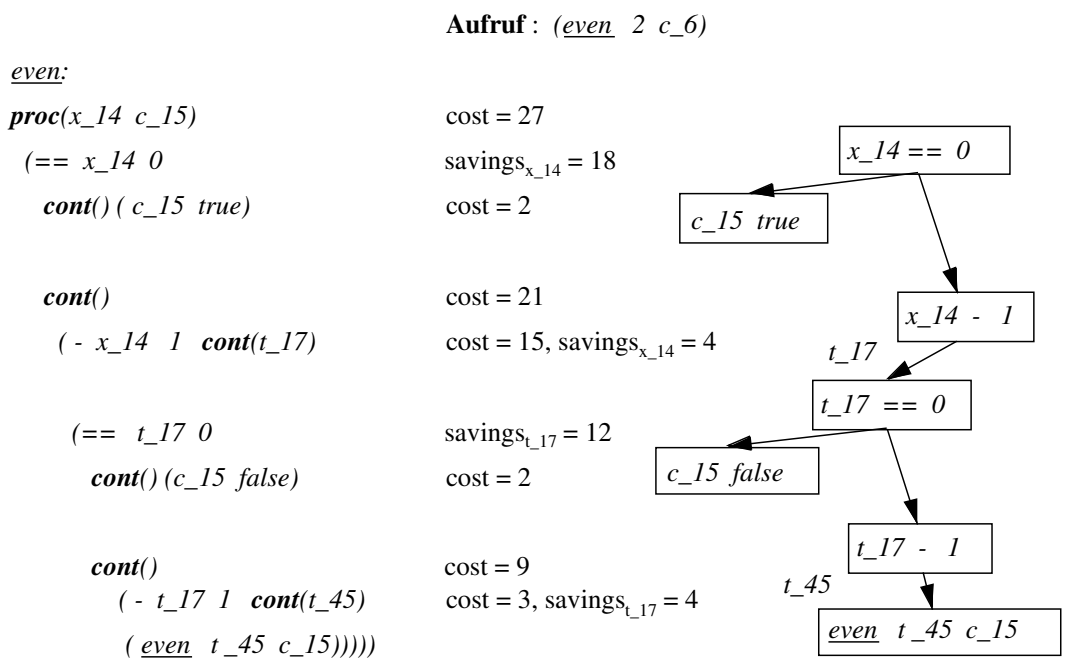



Abbildung 5.4: Beispiel für die Kosten- und *Savings*-Berechnung eines *even*-Aufrufs

aus den Wertargumenten kummuliert (letzte Formel in Abb. 5.3). Der Laufzeitcharakter der *Savings* im Gegensatz zu den Kosten bedingt die Relativierung der schon in den Fortsetzungsverzweigungen für die jeweilige Variable berechneten *Savings*. Diese Relativierung ($\frac{1}{m}$) vermeidet die Berücksichtigung aller Codeverzweigungen, da zur Laufzeit immer eine davon ausgewählt wird.

Außerdem werden für die Berechnung der potentiellen *Savings* an Aufrufknoten elementarer Operationen folgende Auswirkungen des *Inlining* berücksichtigt:

1. Die elementare Operation wird eliminiert, also werden ihre Ausführungskosten *cost(p)* eingespart.

2. Der Zugriff auf die $n$ Argumente der elementaren Operation wird ausfallen ($+n$).

3. Eine neue Variable für das Ergebnis der Evaluation wird alloziert. Dies verringert die *Savings* um eine Einheit (*-1*).

4. Alle Fortsetzungen bis auf die Ausgewählte werden eliminiert. In der Formel wird dazu der Term $\frac{m-1}{m} * \sum_{i=1}^{m} cost(val_i^c)$ verwendet, welcher die Wahrscheinlichkeit zur Eliminierung der $m-1$ Fortsetzungen modellieren soll.

Der Multiplikationsfaktor $\frac{|val_1^t \ldots val_n^t|_v}{\sum_{t \in Temp} |val_1^t \ldots val_n^t|_t}$ drückt den Beitrag der jeweiligen Variable, für welche die *Savings* zu bestimmen sind, zur Auswertung der elementaren Operation aus (im Nenner ist die Anzahl ihrer Referenzen in der Argumentliste, im Zähler die Anzahl der Variablen in der Argumentliste überhaupt).

Während des Aufwärtsganges durch den TML-Baum in der Expansionsphase werden nach den Formeln aus Abb. 5.3 die potentiellen *Savings* aller Variablen festgelegt. Im konkreten *Inlining*-Fall liegt dann diese Information vor, so daß die an Literalen gebundenen Variablen mit ihren *Savings* die geschätzten Reduktionen des Funktionsrumpfes angeben und in die *Inlining*-Entscheidungsregel einfließen.

Falls formale Parameter an Lambda-Abstraktionen gebunden werden, d.h. bei einem Aufruf mit Funktionen als Argumente, können weitere Codevereinfachungen erst nach einer zweiten Stufe, nachdem auch diese Funktionen per *Inlining* substituiert worden sind, vollbracht werden. Auf jeden Fall ist es wichtig, dem Optimierer einen Hinweis auf die konkrete *Inlining*-Situation bei Aufrufen von Funktionen höherer Ordnung zu geben. Aus diesem Grund wird für jedes Lambda-Argument eines Funktionsaufrufs ein zusätzlicher Term zu den jeweiligen *Savings* addiert:

**Aufruf:** $(val_0\ val_1 \ldots val_n\ )$
**Umgebung:** $\rho,\ \lambda(v_1 \ldots v_n\ )app/\ val_0,\ abs_i/\ val_i$

$$savings_{v_i}(app) = savings_{v_i}(app)\ +\ |abs_0|_{v_i}\ *\ \frac{optimisticFactor - pessimisticFactor}{2}$$

Es ist naheliegend, diese grobe Schätzung für indirekte Effekte zweiter Stufe in Abhängigkeit von den Optimierungsparametern zu setzen. Abb. 5.4 zeigt ein Beispiel für die Kosten- und *Savings*-Berechnung der *even*-Funktion, wo an jedem Lambda-Knoten die Kosten des Rumpfes und die potentiellen *Savings* des jeweiligen Parameters (*x_14* bzw. *t_17*) angegeben sind.

Der letzte Term der linken Seite der Ungleichung in Abb. 5.1 auf Seite 71 gehört auch zu den indirekten Effekten des *Inlining* und soll die Wahrscheinlichkeit ausdrücken, mit der die eingefügte Abstraktionsdefinition aus dem Code eliminiert werden kann. Falls eine Funktion nur einmal aufgerufen wird, neutralisiert dieser Term selbst die Kosten der Abstraktion, so daß *Inlining* in diesem Fall garantiert vorgenommen wird, was genau der Beta-Kontraktion (s. Abschnitt 5.2.1) entspricht. Wenn die Funktion zweimal aufgerufen wird, besagt der Term, daß durch ein *Inlining* die Hälfte der Abstraktionsdefinition verschwinden würde (die nächste *Inlining*-Situation würde dann dem Beta-Kontraktions-Fall entsprechen), und so weiter.

Die heuristische Entscheidungsregel von Abb. 5.1 begrenzt eine unkontrollierte Codeexpansion durch die linke Seite der Ungleichung und sorgt durch die rechte Seite für die schnelle

Terminierung durch die Erhöhung der Wirkung des negativen *pessimisticFactor* mit jedem weiteren Optimiererzyklus. Die Erhöhung des optimistischen Faktors für rekursive bzw. nicht rekursive Funktionen hingegen führt zur Kompensation der Kosten des jeweiligen Funktionsrumpfes. Dies wirkt sich auf die Entscheidung zugunsten einer Expansion aus (s. Abschnitt 7.3).

Es ist zu beachten, daß in der *Inlining*-Regel die rekursiven Funktionen nur in Bezug auf ihren unterschiedlichen *optimisticFactor* erfaßt werden, obwohl es zu erwarten ist, daß genau diese Funktionen das größte Problem für die Terminierung und Effizienz des Expansionsalgorithmus darstellen. Einer unendlichen Expansion rekursiver Funktionen wird von dem Bottom-Up-Algorithmus der Expansionsphase vorgebeugt. Bei der Traversierung des Baumes wird registriert, innerhalb von welchen rekursiven Funktionsdefinitionen sich die zu behandelnde *Inlining*-Situation befindet. Rekursive Aufrufe innerhalb der jeweiligen Lambda-Abstraktionen werden nicht expandiert. Transformationen, die auf solchen Expansionen beruhen (z.B. Ausrollen von Schleifen), haben in den meisten Fällen keinen bedeutenden indirekten Effekt, so daß ein *Inlining* kaum Performanzvorteile bringen würde. In manchen Optimierern (z.B. in [Chambers 92]) wird die Datenflußanalyse für Schleifen durch ein Schleifen-Ausrollen unterstützt. Beim TL-Optimierer gibt jedoch die TML-Repräsentation einfachere Möglichkeiten zur effizienten Behandlung solcher Analysen (z.B. durch Erweiterung von TML-Knoten um Felder für die Variablendefinitions- und Variablenverwendungsmengen, s. Abschnitt 6.3).

Durch diese Methode zur Behandlung der rekursiven Funktionsaufrufe finden Expansionen vor allem an denjenigen Stellen des Programms statt, wo sich die größten Chancen für eine künftige Codevereinfachung bieten – nämlich bei Funktionsaufrufen, deren Argumente bekannt (Literale oder Lambda-Abstraktionen) sind. Dieses ist eher außerhalb als innerhalb rekursiver Abstraktionen möglich. Das Verfahren garantiert auf der einen Seite einen effizienten Laufzeitcode, auf der anderen eine schnelle Terminierung des Expansionsalgorithmus.

#### 5.3.2.2 Der Expansionsalgorithmus

Die Expansionsphase des Optimierers bekommt als Eingabe einen vollständig reduzierten TML-Baum und transformiert ihn, indem die geeigneten Funktionen anstelle ihrer Aufrufe kopiert werden. Anschließend wird die Reduktion erneut aufgerufen. Dieser Zyklus wird so lange wiederholt, bis kein *Inlining* mehr möglich ist. Damit spielt die Expansionsphase und speziell die Entscheidungsregel für das *Inlining* (s. Abb. 5.1 auf Seite 71) die Rolle des Abbruchkriteriums für den Optimierer.

Auch in dieser Phase traversiert der Algorithmus den TML-Baum durch einen rekursiven Abstieg, bei dem für jeden Knotentyp die entsprechenden Manipulationen definiert sind. Eine formale Beschreibung des Expansionsalgorithmus ist im Anhang B.3 angegeben. Ähnlich wie bei der Reduktionsphase wird im folgenden lediglich auf die wichtigsten Aspekte des Algorithmus eingegangen.

Die Regel *expand* definiert die Aktionen an allen möglichen Knotenarten im TML-Baum. Auf Literalknoten, die eine Objektspeicherreferenz (*oid*) darstellen, wird die Regel *unloadOid* angewendet, die die reflektive PTML-nach-TML-Umwandlung enthält. Hier läßt sich der Unterschied zwischen der statischen und dynamischen Phase des Optimierers erkennen. Während in der statischen Phase globale TML-Variablen auf die dem aktuellen Code fremden Objekte hinweisen, werden sie nach dem Binden durch die jeweiligen Objektspeicherreferenzen ersetzt, die somit dem Optimierer die Möglichkeit gewähren, auch den Code der fremden Objekte zu untersuchen. Also kommt die Regel *unloadOid* nur bei der dynamischen Optimierung zum Zuge.

Die Regel *unloadOid* bereitet eine Objektreferenz auf das *Inlining* vor und leitet die geschachtelte Optimierung ein. Sie wird für *oid*s aufgerufen, die Objektstrukturen (*arrays*) und Funktionsabschlüssen (*closures*) referenzieren. Dabei wird unterschieden, ob das *oid* schon bekannt ist oder nicht. Das Prädikat *isKnown* wird durch eine Buchführung aller bis zu diesem Moment von *unloadOid* ausgepackten Speicherobjekte festgelegt. Bei den unbekannten Speicherstrukturen (*arrays*) wird die Regel *unloadOid* rekursiv auf jedes Feldelement angewendet. Bei den *oid*s, die unbekannte Funktionsabschlüsse referenzieren, wird zunächst die PTML-nach-TML-Konvertierung vorgenommen. Die Voraussetzung für diese reflektive Umwandlung ist die Existenz einer PTML-Repräsentation für das Funktionsobjekt im Objektspeicher (d.h., daß bei seiner separaten Übersetzung die PTML-Codegenerierung gewählt worden ist).

Auf eine aus dem Objektspeicher geholte Funktion wird dann die Regel *nestedOptimize* angewendet, welche in Abhängigkeit davon, ob das gespeicherte Funktionsobjekt schon einmal dynamisch optimiert worden ist oder nicht, (erkennbar an dem *unknownRefs*-Feld der persistenten PTML-Struktur, s. Abb. 3.5 auf Seite 25), entsprechend einmal den TML-Baum reduziert (Applikation von *reduce*, s. Abschnitt 5.3.1), bzw. die Reduktion und Expansion abwechselnd aufruft, bis die Lambda-Abstraktion vollständig optimiert ist (s. Regel *nestedOptimizeLoop*). Eine Reduktion von schon optimierten, aus dem Objektspeicher geholten Lambda-Abstraktionen ist erforderlich, da der TML-Code durch die Ausnahme-Konvertierung und bei der PTML-Generierung (s. Abschnitt 4.2.4) modifiziert worden ist. Nähere Erläuterungen hierzu finden sich in Abschnitt 5.4.

An einem Applikationsknoten traversiert die *expand*-Regel zunächst die Argumente, inclusive der Fortsetzungsargumente (nach dem Bottom-Up-Prinzip). Bei der Operation $Y$ gibt es einen Wechsel in der Reihenfolge der Anwendung von *expand* auf die für die wechselseitig rekursiven Funktionen stehenden Lambda-Abstraktionen. Der Grund dafür ist, daß gemäß der Überlegungen aus Abschnitt 5.3.2.1 rekursive Aufrufe innerhalb der jeweiligen Abstraktionen nicht expandiert werden. Die Kosten und *Savings* einer solchen rekursiven Abstraktion sind während ihrer Bearbeitung noch nicht berechnet. Bei der sequentiellen Traversierung von $Y$-Funktionen können schon berechnete Kosten und *Savings* durchaus verwendet werden, d.h., daß die vorderen Funktionen in der Liste der wechselseitig rekursiven Funktionen in den hinteren bereits expandiert werden können. Um eine Symmetrie einigermaßen herzu-

stellen, wird durch die Regel *reverse* die Bearbeitungsreihenfolge der wechselseitig rekursiven Lambda-Abstraktionen bei jedem Optimiererzyklus umgekehrt. Als letzte wird bei $Y$ die normale Fortsetzung expandiert, nachdem bereits die rekursiven Lambda-Abstraktionen behandelt worden sind.

Bei der Expansion einer Lambda-Applikation werden zunächst die Wert- und Fortsetzungsargumente nach dem Bottom-Up-Prinzip traversiert und anschließend die Regel *expand* auf die Lambda-Abstraktion in Funktionsposition angewendet. Anschließend werden die unbenutzten Variablen bzw. Argumente durch *eliminateArgs* eliminiert.

Kandidaten für das *Inlining* sind die Variablen- bzw. Literal- (*oid*-) Applikationen. In beiden Fällen wird die Regel *inlineCall* nach der Expansion der Argumente aufgerufen, wobei davor das *oid*-Literal durch *unloadOid* mit der zugehörigen Lambda-Abstraktion durch die PTML-nach-TML-Konvertierung versehen wird. Wie es schon erwähnt wurde, kommt letzteres nur während der dynamischen Optimierung vor.

Die Regel *inlineCall* enthält die Applikation der Grundregel *inline*, des eigentlichen *Inlining*. Für Variablen an Funktionspositionen wird zunächst eine Substitution versucht. Falls für die Funktionsvariable ein *oid*-Literal steht, wird seine Abstraktion über *unloadOid* bekannt gemacht. Damit kommt man zum Fall des Literal-*Inlining*.

Nun kann das eigentliche *Inlining* (Grundregel *inline*) erfolgen, falls die erforderlichen Voraussetzungen gegeben sind (die in Abschnitt 5.3.2.1 erläuterte Entscheidungsungleichung soll gelten).

Nach der erfolgreichen Expansion der Funktion wird mit *nestedInline* eine geschachtelte Expansion im substituierten Funktionsrumpf durchgeführt. Auf diese Weise werden die sich im Rumpf befindenden Aufrufe expandiert, ohne dazu auf die nächste Reduktionsphase zu warten. Gute Kandidaten für die geschachtelte Expansion sind Funktionen, die andere Funktionsabstraktionen als Argumente erhalten und diese aufrufen. In diesem Fall wird nach Applikation der *bind*-Regel auf die neu (nach dem *Inlining*) entstandene Lambda-Applikation in die Lambda-Abstraktion eingestiegen und auf einer tieferen Ebene expandiert.

Trotz der teueren Kopieroperationen während der Expansionsphase bleibt die Anzahl der traversierten TML-Knoten gleich der Kardinalität des resultierten Baumes, da ein und derselbe Knoten auf keinen Fall zweimal besucht wird.

## 5.4 Laufzeit- vs. Optimierereffizienz

Die Komplexität des Optimierungsproblems stammt aus dem Kompromiß zwischen Laufzeiteffizienz des generierten Codes und dem Optimierungsaufwand für seine Erzeugung. Diese beiden Komponenten lassen sich auch in der die Terminierung des TL-Optimierers bestimmenden Entscheidungsregel für das *Inlining* (s. Abb. 5.1 auf Seite 71) erkennen. Durch

die in der Programmierumgebung angebotene Funktion *optimize* (vom Modul *reflect*) hat der Tycoon-Programmierer die Freiheit, die von ihm bestimmten (i.d.R. häufig benutzten) Funktionen dynamisch zu optimieren. In einem persistenten polymorphen System wie Tycoon sind Generatorfunktionen und Datenbankanfragen sehr gute Kandidaten für eine Optimierung nach dem Binden. Beispiele für die Realisation und Optimierung von Anfragen werden in Kapitel 6 angeboten, während die Auswirkungen der dynamischen Optimierung auf Generatoren in Kapitel 7 am Beispiel des TL-Parsergenerators und des TL-Übersetzers selbst gemessen und bewertet werden. Dieser Abschnitt präsentiert zunächst einige Beispiele dafür, wie sich die Optimierungstechniken auf verschiedene TL-Konstrukte auswirken. Im zweiten Teil werden die Maßnahmen zur Steigerung der Effizienz der Optimierung selbst diskutiert.

### 5.4.1 Optimierungsbeispiele

Im folgenden werden einige TL-Konstrukte präsentiert, um die möglichen Optimierungstransformationen an TL-Programmfragmenten zu veranschaulichen.

Diese Demonstration ist folgendermaßen aufgebaut: Zunächst wird der TL-Quelltext angegeben, der aus einem Aufruf der Funktion *optimize* auf eine Funktionsabstraktion besteht, die den zu optimierenden Ausdruck als Rumpf enthält. Darauf folgen die Repräsentationen des aus PTML gewonnenen und des dynamisch optimierten TML-Codes. Anschließend wird auf die eingesetzten Optimierungstechniken hingewiesen.

1. Auswertung konstanter arithmetischer Ausdrücke :

| ***TL:*** | ***TML aus PTML:*** | ***TML optimiert:*** |
|---|---|---|
| *optimize(***fun***()*<br>*1 + {2 * 3});* | *(λ(*_9 +_10)*<br>*(*_9 2 3* ***cont****(t_12)*<br>*(+_10 1 t_12* ***cont****(t_13)*<br>*(cc t_13)))*<br>*<oid 0x006877bc>*<br>*<oid 0x006877dc>)* | *(cc 7)* |

Selbst die elementaren arithmetischen (ganzzahligen) Operationen sind in TL nach dem *add-on*-Ansatz ([Matthes, Schmidt 91]) realisiert. Sie sind in der initialen Umgebung an den konventionellen Zeichen gebunden. Als Funktionen, die dem Übersetzer nicht implizit bekannt sind, können sie erst nach dem Binden expandiert und evaluiert werden.

2. Verwendung globaler Bindungen:

| ***TL:*** | ***TML aus PTML:*** | ***TML optimiert:*** |
|---|---|---|
| *optimize(**fun**() **begin*<br>  ***let*** *gl = 13*<br>  ***let*** *f(x :Int) = gl − x*<br>  *f(7)*<br>***end****);* | *(λ(−_14)*<br>  *(λ(gl_16)*<br>  *(λ(f_17)*<br>  *(f_17  7  **cont**(t_18)*<br>  *(cc  t_18))*<br>  ***proc****(x_19  c_20)*<br>  *(−_14  gl_16  x_19  **cont**(t_21)*<br>  *(c_20  t_21)))*<br>  *13)*<br>*<oid 0x006877cc>)* | *(cc  6)* |

Die globale, unveränderliche Bindung *gl* wird in die Funktion *f* substituiert, was weitere Codevereinfachungen erlaubt. Der Aufruf von *f* wird durch die Abstraktion substituiert. Die anschließende Beta-Reduktion führt zur vollständigen Auswertung. An diesem Beispiel kann man gut die Stellen im TML-Code erkennen, wo die $\eta$-Reduktion in Frage kommt: die Fortsetzungsfunktionen der Aufrufe von *f_17* und *−_14* reichen einfach ihre Parameter (*t_18* bzw. *t_21*) an andere Fortsetzungen weiter. Daher kann darauf die Regel *$\eta$-reduce* angewendet werden. Diese Situation kommt häufig nach der TL-nach-TML-Transformation vor, was auch aus den anderen Beispielen ersichtlich wird.

3. *if*-Konvertierung:

   Die in vielen Übersetzern (*EPIC* [Kessler et al. 86], *Orbit* [Kranz et al. 86], *HARE* [Teodosiu 91]) verwendete *if*-Konvertierung erlaubt eine Vereinfachung von geschachtelten Konditionalen. In Tycoon wird sie auf einer generellen Ebene durch die Applikation der Regeln *case-assign*, *subst* und *fold* auf die Operation '==' realisiert, welche sowohl das *if*- als auch das *case*-Konstrukt implementiert (vgl. Abschnitt 4.3.4). Ein solcher Fall wurde in Abschnitt 5.2 behandelt.

4. Rekursive Funktionen:

| ***TL:*** | ***TML aus PTML:*** | ***TML optimiert:*** |
|---|---|---|
| ***let rec*** *even(x :Int) :Bool =*<br>  *x == 0* ***orif*** *odd(x−1)*<br>***and*** *odd(x :Int) :Bool =*<br>  *x != 0* ***andif*** *even(x−1)*<br>*optimize(**fun**() even(3));* | *(λ(even_27)*<br>  *(even_27  3  **cont**(t_29)*<br>  *(cc  t_29))*<br>*<oid 0x011c0430>)* | *(cc  false)* |

Hier wurde der Aufruf von der rekursiven Funktion *even* mit dem Argument *3* vollständig ausgewertet. Als erstes wurden die wechselseitig rekursiven Abstraktionen von *odd* und *even* in der Tiefe ihrer Aufrufhierarchie optimiert, wobei die *odd*-Funktion im Rumpf von *even* eingesetzt werden konnte. Anschließend wurde an der Aufrufstelle von *even* in zwei aufeinanderfolgenden Optimiererzyklen expandiert, so daß eine vollständige Auswertung erreicht wurde.

Das folgende Beispiel zeigt einen anderen Aspekt der Optimierung von wechselseitig rekursiven Funktionen:

***TL:***

```
let gl = 999
let threeRecs =
 fun() begin
  let rec f(n: Int) :Int =
    f(n-1)
  and g() :Int =
    f(10)
  and t(x :Int) :Int =
    if x==0 then gl
    else g() end
  t(0)
 end
optimize(threeRecs);
```

***TML aus PTML:***

```
(λ(gl_29 ==_30 -_31)
  (Y proc(c_33 t_34 g_35 f_36)
  (c_33
    cont()
      (t_34 0 cont(t_37)
      (cc t_37))
    proc(x_38 c_39)
      (λ(join_40)
        (==_30 x_38 0 cont(t_41)
        (== t_41 true false
          cont() (join_40 gl_29)
          cont()
            (g_35 cont(t_42)
            (join_40 t_42))))
      cont(t_43) (c_39 t_43))
    proc(c_44)
      (f_36 10 cont(t_45)
      (c_44 t_45))
    proc(n_46 c_47)
      (-_31 n_46 1 cont(t_48)
      (f_36 t_48 cont(t_49)
      (c_47 t_49)))))
999
<oid 0x0068771c>
<oid 0x006877cc>)
```

***TML optimiert:***

```
(cc 999)
```

Nach dem *Inlining* des Aufrufs von $t$ und der anschließenden Reduktion leisten die Anwendung der Regel *Y-reduce* und die Evaluation des $Y$-Kombinators den entscheidenden Beitrag zur Evaluation des Ausdrucks.

5. Funktionen höherer Ordnung:

**TL:**

```
let succ(x :Int) =
  x+1
let twice(f(:Int):Int
          a :Int) =
  f(f(a))
optimize(fun()
  twice(succ 9));
```

**TML aus PTML:**

```
(λ(succ_20  twice_21)
  (twice_21  succ_20  9  cont(t_23)
  (cc  t_23))
< oid 0x01337b04 >
<oid 0x01337b18>)
```

**TML optimiert:**

```
(cc  11)
```

Nach der geschachtelten Optimierung von *succ* und *twice* wird das *Inlining* des Aufrufs von *twice* vorgenommen. Das Besondere dabei ist, daß noch in derselben Expansionsphase in den Rumpf von *twice* abgestiegen wird, so daß gleich darauf *succ* expandiert werden kann. Durch das geschachtelte *Inlining* (s. Regel *nestedInline* in Abschnitt 5.3.2.2) wird es ermöglicht, daß alle denkbaren Funktionsexpansionen im ersten Lauf getätigt werden, ohne auf die Reduktionsphase warten zu müssen. Im konkreten Beispiel führt das geschachtelte *Inlining* zur Ersparung eines Optimiererzyklus.

6. Veränderliche Bindungen:

**TL:**

```
let swap(A <:Ok var x,y :A) =
  begin let t = x  x:=y  y:= t end
let var a = 3 and var b = 5
optimize(fun()
  begin swap(:Int a b) a end);
```

**TML aus PTML:**

```
(λ(b_22  a_23  swap_24)
  (swap_24  a_23  0  b_22  0  cont(t_26)
  ([]  a_23  0  cont(t_27)
  (cc  t_27)))
<oid 0x00fc442c>
<oid 0x00fc4420>
<oid 0x00fc440c>)
```

**TML optimiert:**

```
([]  <oid 0x00fc4420>  0  cont(t_58)
([]  <oid 0x00fc442c>  0  cont(t_59)
([]:=  <oid 0x00fc4420>  0  t_59  cont()
([]:=  <oid 0x00fc442c>  0  t_58  cont()
([]  <oid 0x00fc4420>  0  cc)))))
```

Auch bei der Optimierung veränderbarer Werte bzw. Funktionen mit Referenzparametern spielt das *Inlining* eine entscheidende Rolle. Im konkreten Beispiel ist der Aufruf von *swap* expandiert und wegoptimiert worden, so daß nur die elementaren Zugriffsoperationen im Code verbleiben. Man kann erkennen, daß der TL-Optimierer selbst ohne die teuere Datenflußanalyse auch im imperativen Kontext gute Ergebnisse liefern kann.

7. Wertkonstruktoren und Wertselektion:

| ***TL:*** | ***TML aus PTML:*** | ***TML optimiert:*** |
|---|---|---|
| ***let*** *peter :Person =* | *(λ(peter_15 birth_16)* | *(cc 1967)* |
| ***tuple*** *"Peter" 27* ***end*** | *(birth_16 peter_15* ***cont****(t_18)* | |
| ***let*** *birth(p :Person) =* | *(cc t_18))* | |
| *1994 − p.age* | *<oid 0x00f920b0>* | |
| *optimize(****fun****()* | *<oid 0x00f920c4>)* | |
| *birth(peter));* | | |

Die Funktion *birth* enthält einen Zugriff auf das Tupelfeld *age* von *peter*. Nach dem *Inlining* wird festgestellt, daß *peter* ein unveränderbares Tupel darstellt. Folglich wird der Feldzugriff ('[]') ausgeführt.

8. Schleifen:

   Das Beispiel in Abb. 5.5 demonstriert das Ausrollen von Schleifen (*loop unrolling*) in TML. Die Schleife wird in TML als eine rekursive Funktion dargestellt. Nach der Optimierung ihres Rumpfes wird der Schleifeneingang (der Applikationsknoten *(for_27 1)*) zweimal expandiert, so daß in der normalen Fortsetzung des $Y$-Aufrufs der Speicherwert von *sum* zweimal nacheinander inkrementiert wird (implementiert durch die drei aufeinanderfolgenden Aufrufe von '[]', '+' und '[]:='). Nach dem zweimaligen Ausrollen der **for**-Schleife beginnt die „restliche" Schleife im optimierten TML-Code ab dem Index *3*.

9. Polymorphie und Subtypisierung:

| ***TL:*** | ***TML aus PTML:*** | ***TML optimiert:*** |
|---|---|---|
| ***let*** *student =* ***tuple*** | *(λ(student_17 name_18)* | *(cc 1967)* |
| ***let*** *name = "Peter"* | *(birth_18 student_17* | |
| ***let*** *age = 27* | ***cont****(t_20) (cc t_20))* | |
| ***let*** *semester = 7* | *<oid 0x00ed0d8c>* | |
| ***end*** | *<oid 0x00ed0da4>)* | |
| ***let*** *birth(p :Person) =* | | |
| *1994 − p.age* | | |
| *optimize(****fun****()* | | |
| *birth(student));* | | |

Der Wert *student* ist von einem Subtyp von *Person*. Damit tritt der Fall eines Subtyppolymorphismus beim Aufruf der Funktion *birth* auf, deren Signatur *Fun(:Person):Int* ist. In TML sind jedoch diese Beziehungen nicht von Bedeutung, da sich diese Sprache als Lambda-Kalkül erster Ordnung lediglich auf der Ebene der Werte bewegt. Daher sieht die Situation hier nicht anders aus als im 7. Beispiel (Wertkonstruktoren und Wertselektion).

**TL:**

```
let var sum = 0
let succ(x :Int) = x + 1
optimize(fun()
  for i = 1 upto 10 do
    sum := succ(sum)
  end);
```

**TML aus PTML:**

```
(λ(sum_21 succ_22 :=_23)
  (λ(forEnd_25)
    (Y proc(c_26 for_27)
      (c_26
        cont() (for_27 1)                    Schleifeneingang
        cont(i_28)                           Schleifenkopf
          (> i_28 10
            cont()
              (forEnd_25)
            cont()
              ([] sum_21 0 cont(t_29)
              (succ_22 t_29 cont(t_30)
              (:=_23 sum_21 0 t_30 cont(t_31)
              (+ i_28 1 cont(t_32)
              (for_27 t_32))))))))           Iterationsschritt
  cont() (cc nil))
<oid 0x00f24148>
<oid 0x00f24154>
<oid 0x0068772c>)
```

**TML optimiert:**

```
(Y proc(c_26 for_27)
(c_26
  cont()
    ([] <oid 0x00f24148> 0 cont(t_58)        1. Ausrollen
    (+ t_58 1 cont(t_61)
    ([]:= <oid 0x00f24148> 0 t_61 cont()
    ([] <oid 0x00f24148> 0 cont(t_67)        2. Ausrollen
    (+ t_67 1 cont(t_68)
    ([]:= <oid 0x00f24148> 0 t_68 cont()
    (for_27 3)))))))                         Schleifeneingang
  cont(i_28)
    (> i_28 10
      cont() (cc nil)
      cont()
        ([] <oid 0x00f24148> 0 cont(t_29)
        (+ t_29 1 cont(t_30)
        ([]:= <oid 0x00f24148> 0 t_30 cont()
        (+ i_28 1 for_27)))))))
```

Abbildung 5.5: Ausrollen von Schleifen

Die obigen Beispiele zeigen die Einsatzmöglichkeiten des TL-Optimierers in verschiedenen komplizierten TL-Sprachkonstrukten. Sie stellen die Auswirkungen der beschränkten Anzahl von Grundregeln dar, die auf der TML-Ebene definiert sind und die bis zur vollständigen Evaluation von großen TML-Teilbäumen führen können. Auf diese Codetransformationen lassen sich die guten Ergebnisse der Messungen in Kapitel 7 zurückführen.

### 5.4.2 Maßnahmen zur Verbesserung der Effizienz des TL-Optimierers

Eine Steigerung der Laufzeiteffizienz sollte nicht auf Kosten eines unannehmbaren Optimierungsaufwands erreicht werden. Trotz der Möglichkeit zur Auswahl der zu optimierenden Funktionen und Moduln sollte die dynamische Optimierung trotz der potentiell großen Tiefe der geschachtelten Optimierungen nicht einen größeren Zeitanteil als die anderen Komponenten der Übersetzung (Parsing, Typüberprüfung, Codegenerierung) fordern. Wie die Messungen in Abschnitt 7.2 zeigen, konnte dieses Ziel bei der Implementation des TL-Optimierers erreicht werden. Im folgenden werden einige Maßnahmen hierfür dargestellt:

1. Die überwiegend Top-Down-Traversierung in der Reduktionsphase:

   Das im Abschnitt 5.3.1 erläuterte Problem des unterschiedlichen Charakters der Reduktionsregeln bedarf Umkehrungen in der Traversierungsrichtung, damit der TML-Code in einem Lauf vollständig reduziert werden kann. Für die Anzahl der bearbeiteten Knoten ist jedoch von Bedeutung, wie die Traversierung gestartet wird – Top-Down oder Bottom-Up. Im CPS-Fall unterscheiden sich die beiden Methoden an einem Applikationsknoten dadurch, ob die Fortsetzungen vor dem jeweiligen Knoten (Bottom-Up) oder danach (Top-Down) bearbeitet werden.

   Im Gegensatz zu [Gawecki 91] und [Steele 78], wo zuerst mit der Bottom-Up-Traversierung eines Applikationsknotens, mit Ausrichtung auf die Beta-Reduktion, begonnen wird, fängt die Reduktion beim TL-Optimierer Top-Down mit dem Ziel an, als erstes eine frühe Auswertung konstanter Ausdrücke zu ermöglichen. So wird zunächst die Funktionsposition (Lambda-Abstraktion oder Variable) zusammen mit den Wertargumenten reduziert und dann, bevor man in die Fortsetzungen übergeht, eine Reduktion des Knotens selbst versucht (s. Regel *reduce* für die Applikationsknoten im Anhang B.3). Diese Wahl minimiert die Anzahl der traversierten Knoten und kann sogar zu einer Situation führen, in der sie kleiner als die Anfangskardinalität des TML-Baumes ist. Diese Situation wurde im Abschnitt 5.3.1 mit Hilfe eines Beispiels veranschaulicht.

2. Die Bottom-Up-Traversierung des TML-Baumes während der Expansionsphase:

   Bei der Expansion wird das Bottom-Up-Verfahren vorgezogen, da dadurch die Kosten und *Savings* parallel zur Baumtraversierung berechnet werden können. Bei der Bottom-Up-Methode werden die schon einmal expandierten Fortsetzungen nicht mehr bearbeitet, so daß einmal berechnete und an einem Lambda-Knoten gespeicherte Kosten und *Savings* während derselben Expansionsphase nicht mehr verändert werden.

3. Parallele Berechnung der Kosten und *Savings* während der Expansionsphase:

   Zwei Ursachen gestalten die Expansionsphase aufwendiger als die Reduktionsphase: die Kopiersemantik des *Inlining* und die von den Eigenschaften eines ganzen Teilbaumes abhängende *Inlining*-Entscheidung. Die einzige Aufwandreduktion im ersten Fall ist die Ersetzung des Kopierens durch Verschieben (oder Umhängen), falls nur noch ein einziger (letzter) Aufruf zu expandieren ist.

   Bei der Berechnung der Kosten und *Savings* wurden einige Versuche durchgeführt, wobei sich die parallele Berechnung als die effizienteste Alternative durchgesetzt hat. Zwei andere Möglichkeiten in diesem Vergleich waren zum einen die Berechnung pro *Inlining*-Fall und zum anderen die einmalige Bestimmung der Kosten und *Savings* bei der TML-Generierung mit anschließender Korrektur bei jeder Baumänderung an den betroffenen Lambda-Knoten. Das erste Verfahren schied wegen der Notwendigkeit einer wiederholten Traversierung der schon bearbeiteten Teilbäume aus. Bei der zweiten Methode wurde festgestellt, daß die vielseitigen Baummanipulationen während der Reduktionsphase einen immensen Aufwand bei der Kosten- und *Savings*-Korrektur verursachen. Der monotone Bottom-Up-Aufstieg in der Expansionsphase gestattet die Bestimmung der Entscheidungsparameter ohne wiederholte Traversierungen, und zwar in der Phase, wo sie tatsächlich gebraucht werden.

   Die Kosten einer Lambda-Abstraktion werden am Lambda-Knoten selbst gespeichert, während die (potentiellen) *Savings* an jedem Variablenknoten akkumuliert werden. Damit verbleibt der *Inlining*-Entscheidung nur, diese gespeicherten Werte (die Kosten von der einzusetzenden Lambda-Abstraktion und die *Savings* dieser Lambda-Parameter, für die Konstantenargumente im Aufruf stehen) zu vergleichen.

   Der Nachteil der gewählten Alternative besteht darin, daß für die rekursiven Aufrufe innerhalb derselben Funktionsabstraktion die Kosten und *Savings* noch nicht vorliegen. Statt an dieser Stelle die Methode durch eine aufwendige Neuberechnung zu ändern, wurde die Entscheidung getroffen, daß solche Expansionen zu verbieten sind. Eine Diskussion über die wechselseitig rekursiven Funktionen und über ihre gleichberechtigte Behandlung wurde schon in Abschnitt 5.3.2 präsentiert.

4. Persistente Speicherung der Kosten und *Savings* im PTML:

   Die Optimierung endet mit einer Expansionsphase, in der die Kosten und *Savings* endgültig berechnet worden sind. Bei der inhärenten Persistenz des Tycoon-Systems ist es naheliegend, diese Informationen für eine weitere Benutzung im Funktionsabschluß zu behalten (s. Abschnitt 3.3). Dadurch liefert eine solche Funktion bei einem späteren Versuch, sie während der dynamischen Optimierung zu expandieren, gleich die für die *Inlining*-Entscheidung notwendigen Informationen. Schwierigkeiten ergeben sich aus dem störenden Einfluß der anderen im Reflektionszyklus liegenden *Back-End*-Komponenten des TL-Übersetzers: der Ausnahme-Konvertierung und der Codegenerierung. Daher wird bei der geschachtelten Optimierung (Regel

*nestedOptimize*, s. Abschnitt 5.3.1.2) immer einmal die Reduktionsphase auf die aus PTML gewonnenen Lambda-Abstraktionen aufgerufen. Sie korrigiert die durch die Ausnahme-Konvertierung veränderten Kosten und *Savings* und entfernt die globalen Variablennamen, die für Debuggingzwecke bei der PTML-Codegenerierung beibehalten werden.

In [Dean, Chambers 93] wird ein anderes Verfahren der Nutzung schon berechneter Ergebnisse verwendet. Für den SELF-Übersetzer wird eine Datenbank mit den bisherigen Funktionsaufrufen und mit den Entscheidungen pro oder kontra *Inlining* verwaltet. Die letzteren werden durch Versuche für bestimmte Argumenttypen pro Funktion festgelegt und in der Datenbank gespeichert. Im Endeffekt ist das Ziel in beiden Systemen dasselbe: die Reduktion des Aufwands bei der *Inlining*-Entscheidung, wobei das Problem im Tycoon-System einfacher und billiger (keine Datenbankverwaltung) ohne Beeinträchtigung der Laufzeit gelöst wird (in [Dean, Chambers 93] werden 50 % Verbesserung der Übersetzungszeit mit 15 % Einbuße der Performanz des gesamten Auswertungszyklus erreicht).

5. Unterscheidung zwischen optimierten und nicht optimierten Funktionsabschlüssen:

   Die Chancen für die Expansion von Funktionen, die aus dem Objektspeicher durch die PTML-nach-TML-Transformation restauriert worden sind, werden bedeutend erhöht, wenn diese voroptimiert werden und erst dann die *Inlining*-Entscheidung getroffen wird. Diese geschachtelte Optimierung legt die Regel *nestedOptimize* fest (s. Abschnitt 5.3.2.2).

   Die Persistenz von Tycoon wird vom TL-Optimierer auch unter einem anderen Aspekt ausgenutzt. Für jede Funktionsabstraktion wird im PTML-Code die Anzahl der globalen Referenzen (im Feld *unknownRefs*, s. Abschnitt 3.3) gespeichert. Daraus kann der dynamische Optimierer erkennen, ob die geschachtelte Optimierung (*nestedOptimizeLoop*) einzusetzen ist oder nicht. Wenn es keine globalen Referenzen gibt, bedeutet es, daß die Funktion entweder solche überhaupt nicht enthält oder daß sie schon einmal dynamisch optimiert worden ist. Die Existenz solcher globalen Referenzen wird jedoch nur während der Optimierung notiert. Um eine nicht optimierte Funktion zu kennzeichnen, wird das Feld *unknownRefs* in ihrem Funktionsabschluß zu einem Wert $\neq 0$ initialisiert.

6. Minimierung der Optimiererzyklen durch geschachteltes *Inlining*:

   Innerhalb eines Optimiererzyklus erzeugt die Reduktionsphase einen vollständig reduzierten TML-Baum. Die Rolle der Expansionsphase ist es, in diesem Baum die möglichen *Inlinings* zu verwirklichen, wobei in der darauffolgenden Reduktionsphase des nächsten Zyklus die sich daraus ergebenden Reduktionen durchgeführt werden.

   Die im Abschnitt 5.3.2.2 betrachtete Regel *nestedInline* ermöglicht, daß auf die *Inlinings* der sich innerhalb einer expandierten Funktion befindenden Funktionsaufrufe

nicht einen ganzen Zyklus lang (Expansion-Reduktion-Expansion) gewartet werden muß. Falls einer expandierten Funktion weitere Lambda-Abstraktionen als Argumente übergeben werden, wird noch während derselben Expansionsphase eine Traversierung der eingesetzten Funktionsrumpfkopie durchgeführt. Das ist der Grund dafür, daß im Beispiel für die Funktionen höherer Ordnung im Abschnitt 5.4.1 die Funktion *twice* in zwei statt drei Optimiererzyklen evaluiert wird.

Eine weitere Maßnahme zur Reduktion der Optimierungszeit in dieser Hinsicht sind zusätzliche Expansionsbedingungen für den Variablen- bzw. *oid*-Funktionsaufruf, die neben der allgemeinen Entscheidungsregel (Abb. 5.1 auf Seite 71) berücksichtigt werden. Einerseits betrifft das die Buchführung über die rekursiven Funktionsabstraktionen, innerhalb denen man sich momentan befindet und für deren Aufrufe das *Inlining* verboten ist. Andererseits werden Variablen-Aufrufe von Fortsetzungsfunktionen auf der obersten Schachtelungsebene der Expansionsphase nicht substituiert. Die Begründung dieser Entscheidung ist die Tatsache, daß dadurch lediglich ein Sprung zur Laufzeit (Fortsetzungen bezeichnen im Maschinencode nämlich Sprünge) auf Kosten aufwendiger Kopierung von manchmal immensen Teilbäumen erspart wird. In den weiteren Schachtelungsstufen jedoch spielt das *Inlining* von Fortsetzungen eine entscheidende Rolle für die Beta-Reduktion, da sie als Argumente im expandierten Funktionsrumpf substituiert werden und zu erheblichen Codevereinfachungen führen können.

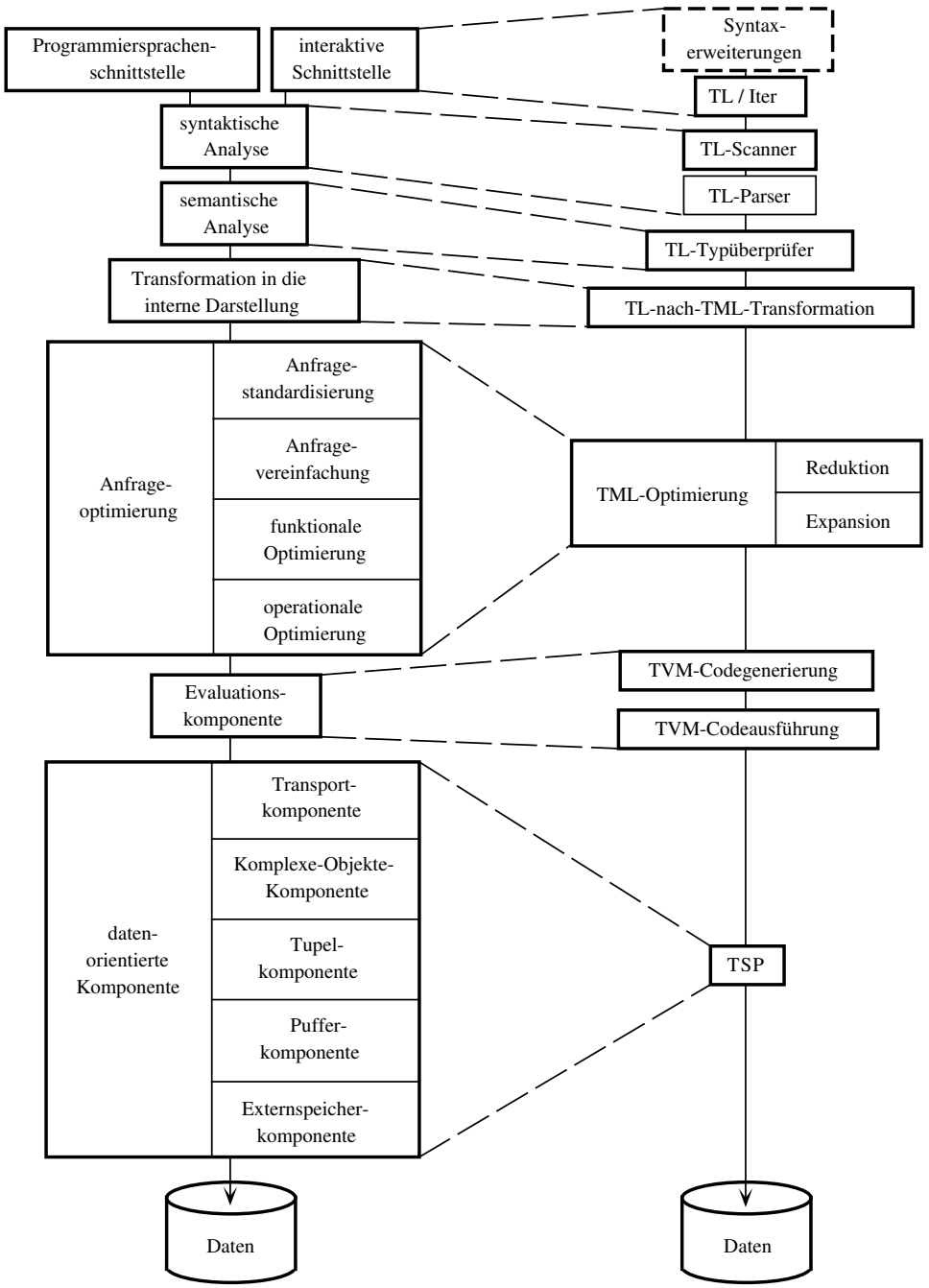
# 6 Datenbankanfragen im Tycoon-System

Innerhalb des Tycoon-Projekts soll die universelle Programmiersprache TL auch als Datenbankprogrammiersprache eingesetzt werden. Dies wird insbesondere durch Eigenschaften wie Persistenz, Polymorphie und Verwendung von Funktionen höherer Ordnung ermöglicht. Dabei agiert TL sowohl als eine Datenbankdefinitionssprache als auch als eine Datenbankmanipulationssprache, die zusätzlich den Vorteil einer statischen Typisierung besitzt.

In diesem Kapitel werden die Auswirkungen der beschriebenen Optimierungsmethoden auf in TL formulierte Datenbankanfragen analysiert. Zunächst wird eine Analogie zwischen der Architektur eines allgemeinen Datenbankanfrageevaluators und des optimierenden TL-Übersetzers gezogen (Abschnitt 6.1). In Abschnitt 6.2 wird die Modulschnittstelle *Iter* als ein generisches Tycoon-Werkzeug zur Iterationsabstraktion vorgestellt. Abschnitt 6.3 bietet eine Beschreibung und Bewertung der Auswirkungen der allgemeinen TL-Optimierung auf der Grundlage einer Sammlung von algebraischen Anfragetransformationen an. Außerdem werden Ideen für weitere Optimierungen skizziert, die bessere Ergebnisse bei der Anfrageevaluation liefern könnten.

## 6.1 Das allgemeine Modell eines Anfrageevaluators und der optimierende TL-Übersetzer

Datenbankanfragen besitzen ein hohes Maß an Abstraktion und Deklarativität, die ihre Evaluation über mehrere Zwischenrepräsentationen bedingen. Insbesondere gilt das für die Anfrageoptimierung auf den verschiedenen Ebenen: auf der logischen, funktionalen, operationalen bzw. physikalischen Ebene. Trotz der Vielzahl an Datenbanksystemen gehorchen sie einem allgemeinen Ansatz bei der Anfrageevaluation, der in Abb. 6.1.a dargestellt ist. Die dort angegebene Referenzarchitektur umfaßt sämtliche Komponenten, die im Rahmen der Anfrageauswertung involviert sind, und dient dem allgemeinen Überblick über die Phasen, die eine Anfrage von ihrer Formulierung bis zu ihrer effizienten Ausführung in einem Datenbanksystem (DBMS) durchläuft (vgl. [Mielinski 91]).




a) Anfrageevaluator in DBMS (nach [Mielinski 91])

b) Tycoon

Abbildung 6.1: Anfrageevaluation vs. TL-Übersetzer

Neben der Referenzarchitektur des allgemeinen Datenbankanfrageevaluators (Abb. 6.1.a) ist in Abb. 6.1.b die entsprechend aufgegliederte Architektur des TL-Übersetzers mit den für ihn relevanten Tycoon-Komponenten dargestellt. Aufgrund vieler Gemeinsamkeiten läßt sich eine Analogie zwischen beiden Architekturen und damit zwischen einem DBMS und dem Tycoon-System ziehen. Diese Analogie sollte nicht überraschen, da eine der bedeutendsten Anwendungen eines persistenten Systems gerade im Bereich der datenintensiven Programmierung liegt. Im folgenden werden die einzelnen Komponenten der Referenzarchitektur des Anfrageevaluators (nach [Mielinski 91]) zusammen mit ihren äquivalenten Tycoon-Elementen kurz diskutiert:

**Programmiersprachen-** und **interaktive Schnittstelle:** Die Schnittstellen auf der höchsten Ebene der DBMS-Architektur bieten zum einen die Möglichkeit zur Einbettung der Datenbanksprache in einer Wirtssprache (z.B. C), zum anderen die Möglichkeit zur direkten Kommunikation mit der Datenbank über *ad hoc* Anfragen. Da das Tycoon-System durch sein einheitliches Bindungs-, Benennungs- und Typisierungskonzept den konzeptionellen Bruch bei der Einbindung von zwei verschiedenen Sprachen vermeidet, vertritt die Systemsprache TL selbst diese beiden Schnittstellen. Die interaktive Programmierumgebung und die funktionale Vollständigkeit von TL erlauben sowohl die *ad hoc* Anfragenformulierung als auch komplizierte Datenmanipulationen, -berechnungen und -austausch zwischen verschiedenen Servern.

**Syntaktische Analyse:** Bei der syntaktischen Analyse wird ein linearer Anfrageausdruck in eine geeignete Repräsentation, den *Syntaxbaum*, überführt und dort auf mögliche Fehler überprüft. Dieser Komponente entsprechen auf Tycoon-Seite der TL-Parser und der dazu gehörige Scanner. Beide können anhand benutzerdefinierter Syntaxregeln erzeugt werden, so daß verschiedene Syntaxerweiterungen möglich sind ([Schröder 93], [Juhacz 94]), die eine bequeme deklarative Benutzerschnittstelle anbieten.

**Semantische Analyse:** In dieser Phase werden die Variablen- und Typbindungen in der Datenbank durchgeführt, sowie die Sichtbarkeitsbereiche und die Typkorrektheit überprüft. Diese Funktionalität wird vom TL-Typüberprüfer übernommen, der für die statische Analyse von Bindungen und Typen in der *Front-End*-Phase des TL-Übersetzers zuständig ist.

**Transformation in die interne Darstellung:** Obwohl der abstrakte Syntaxbaum der syntaktischen und semantischen Analyse der Datenbankanfragen sehr gut dient, ist er für die Anfrageoptimierung und -evaluation nicht geeignet. Aus diesem Grund wird ein *Anfragebaum* erzeugt, der zum einen Knoten zum Zweck der funktionalen Optimierung (Quantoren-, boolesche, Vergleichs-, Funktionsknoten, etc.), und zum anderen Knoten zum Zweck der operationalen Optimierung bzw. der Erstellung eines Zugriffsplans (Datenzugriffs-, *Join*-, Entscheidungsknoten, etc.) enthält. In Tycoon übernimmt diese Rolle der TML-Baum, der eine viel allgemeinere Programmdarstellung anbietet. Die in der *Front-End*-Phase des TL-Übersetzers stattfindende TL-nach-

TML-Transformation ist das genaue Abbild der jeweiligen Komponente der DBMS-Referenzarchitektur.

**Anfrageoptimierung:** Auf der Grunglage des erzeugten Anfragebaumes wird die Optimierung der Datenbankanfrage vorgenommen. Üblicherweise werden zunächst die Datenbankfunktionen expandiert, wonach die Anfragen in eine standardisierte und vereinfachte Struktur transformiert werden. Darauf werden die algebraischen Regeln der **funktionalen Optimierung** und die *Join*-Verfahren- und Zugriffsregeln der **operationalen Optimierung** angewendet. Zu der ersten Optimierungsphase gehören Transformationen wie die Zusammenfassung von Projektionen und Selektionen und Verschiebung von selektiven Operationen (Projektion, Selektion) vor konstruktive Operationen (*Join*, Kartesisches Produkt). Zur operationalen Optimierung werden globale Transformationen in Bezug auf die *Join*-Reihenfolge (z.B. *Sort-Merge-Join*, *Semi-Join* oder *Nested-Loop-Join*) und auf die Auswahl der Zugriffspfade zu den Daten durchgeführt. Die Transformationen hier finden unter aktiver Ausnutzung von semantischen Informationen und Statistiken über die Daten statt.

Obwohl der TL-Optimierer auf der allgemeinen Ebene der Codereduktion und -expansion agiert, existieren viele Analogien zu dem Anfrageoptimierer, sowie auch Möglichkeiten zur weiteren Annäherung des allgemeinen Optimierungsansatzes in persistenten Systemen an die anfragebezogene Optimierung in Datenbanksystemen. Die bedeutendste Eigenschaft des TL-Optimierers in dieser Hinsicht ist sein dynamischer Charakter, der die Grundlage für seinen Einsatz in der Anfrageauswertung schafft. In Abschnitt 6.3 werden eine Reihe von Regeln der Prädikatevaluation und -vereinfachung demonstriert, die durch die Reduktionsregeln abgedeckt werden. Die Expansionsphase des TL-Optimierers erlaubt das *Inlining* von (Datenbank-)Funktionen, die einen Effizienzvorteil zu bringen versprechen. Durch das Zusammenspiel der beiden Optimierungsphasen sind auch manche der Transformationen der funktionalen Optimierung möglich (z.B. die Zusammenfassung von selektiven Funktionen, soweit sie als *Inlining*-Kandidaten substituiert und ausgewertet werden). Andere Transformationen, wie die Verschiebung von selektiven vor konstruktive Operationen lassen sich durch weitere Standardmethoden der Optimierung wie Code-Verschiebung (*hoisting*) und Eliminierung gemeinsamer Teilausdrücke implementieren. Ideen zu den betreffenden Algorithmen sind in Abschnitt 6.3 zu finden.

Viel komplizierter sind die Fragen der operationalen Optimierung, bei denen Entscheidungen auf der logischen Ebene anhand des Datenbankschemas und unter Ausnutzung von Statistiken getroffen werden. Aus diesem Grund wäre es denkbar, diese Regeln auf einer höheren Ebene, z.B. auf der Ebene der Syntaxerweiterungen, zu definieren und zu implementieren.

**Evaluationskomponente:** Sie erhält den optimierten Zugriffsplan von der operationalen Optimierung und führt ihn aus. Dieser Komponente entspricht im Tycoon-System die Codegenerierungsphase und die Ausführung des daraus entstammenden TVM-Codes.

**Datenorientierte Systemkomponenten:** Hier werden die in verteilten Systemen relevante **Transportkomponente**, die **Komplexe-Objekte-Komponente**, die von der konkreten Struktur der komplexen Objekte abstrahiert, die für die Tupelzugriffe zuständige **Tupelkomponente**, die den Zugriff auf Dateien und Blöcke steuernde **Pufferkomponente** und die den direkten Speicherzugriff ausführende **Externspeicherkomponente** zusammengefaßt. Diese Komponenten erlauben den physikalischen Zugriff auf die in sekundären Speichern liegenden Daten und bieten der Evaluationskomponente eine Abstraktion von diesen speicherbezogenen Aufgaben an. Die direkte Analogie in Tycoon stellt hier das TSP-Protokoll dar, das die Zugriffs-, Verteilungs-, Sperr- und Speicheroptimierungsfunktionalität auf unterschiedlichen Objektspeicherplattformen von den höheren Schichten abkapselt.

Aufgrund der gezogenen Parallelen zwischen der allgemeinen Architektur eines DBMS (in Bezug auf die Anfrageevaluation) und dem Aufbau des TL-Übersetzers ist zu erwarten, daß das Tycoon-System eine geeignete Umgebung zur Lösung von datenintensiven Aufgaben anbietet. Daher ist ein Interesse an den Optimierungsvorgängen bei der Anfrageevaluation in Tycoon begründet. Vor ihrer konkreten Analyse wird ein Werkzeug zur Iterationsabstraktion vorgestellt, daß eine funktionale Schnittstelle zur Anfrageformulierung anbietet.

## 6.2 Die Modulschnittstelle Iter: ein Mechanismus zur Iterationsabstraktion

Der *add-on*-Ansatz [Matthes, Schmidt 91] des Tycoon-Systems erlaubt die erweiterbare Definition und Verwendung von Massendatentypen (*bulk data types*). Mehrere Moduln aus der Tycoon-Standardbibliothek bieten die abstrakten Typdefinitionen sowie auch polymorphe Generierungs- und spezifische Manipulationsfunktionen für Mengen, Listen, Hash-Tabellen und andere Massendatenstrukturen an.

Der bedeutendste Nachteil monomorpher Typsysteme besteht in der Notwendigkeit zur separaten Definition von Iteratorfunktionen für jeden einzelnen Massendatentyp. Zum einen erhöht dieses die Komplexität des Systems (besonders, wenn die Massendatentypen nach dem *built-in*-Ansatz in der Sprache integriert sind), zum anderen erlaubt es nicht die Formulierung von generischen Anfragen, die für verschiedene Massendaten die gleiche Form aufweisen. Als eine polymorphe Sprache mit Funktionen höherer Ordnung in einer persistenten Umgebung bietet TL eine elegante Möglichkeit, generische Anfragen auf komplexe Objekte zu formulieren. Der Kernpunkt stellt der Modul *iter* (mit seiner Schnittstelle *Iter*) dar, der die Funktionalität einer Iterationsabstraktion enthält und dem Benutzer ein einheitliches Werkzeug zur Verarbeitung von Massendaten beliebiger Struktur liefert.

Das Prinzip der Iterationsabstraktion ist in Abb. 6.2 dargestellt (s. [Niederee 92]). Jede Massendatentypschnittstelle bietet eine Funktion *elements*, die die Konvertierung in den

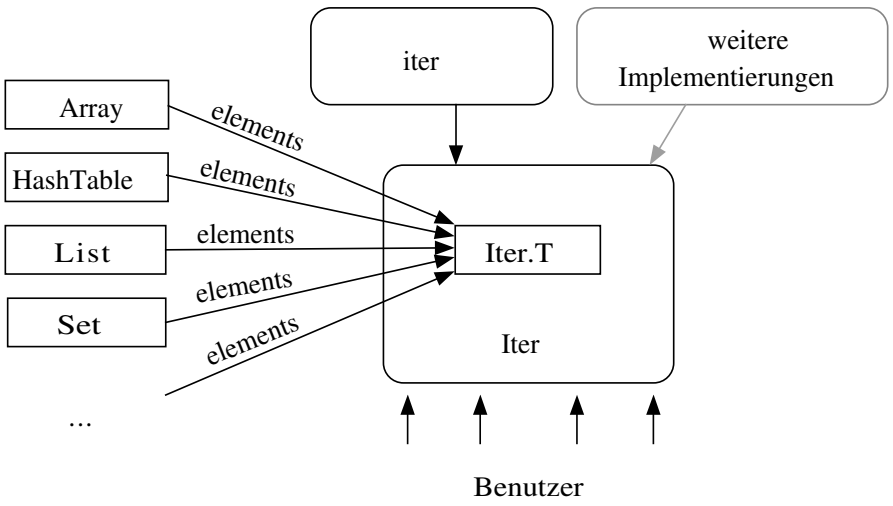



Abbildung 6.2: Massendaten und Iterationsabstraktion in Tycoon (nach [Niederee 92])

Iteratorrepräsentationstyp vollbringt. Auf diesem Typ sind die üblichen Navigations-, Selektions-, Konstruktions- und Manipulationsoperationen definiert. Sie beruhen auf folgenden drei Komponenten des Typs *Iter.T*:

**empty** bezeichnet, ob die Struktur leer ist

**get** gibt das aktuelle Element zurück

**next** springt zum nächsten Element in der Struktur hin[1]

Um die Verwendung der Schnittstelle zu veranschaulichen, wird im folgenden ein kleines Beispiel vorgestellt. Es basiert auf der Datenbank, deren Schema in Abb. 6.3 in TL formuliert ist und die auch im nächsten Abschnitt verwendet wird.

**Ein Anfragebeispiel:** Welche Produkte wiegen mehr als 12 Einheiten?

[1] Ein großer Nachteil dieser Iteratorimplementation ist die Notwendigkeit von drei Funktionsabschlüssen für jedes Element in der Datenstruktur. Dieses widerspiegelt sich sowohl in einem erhöhten Speicherbedarf als auch in Effizienzeinbuße beim Datenzugriff. Aus diesem Grund wird in der gegenwärtigen Tycoon-Entwicklung an einer Iteratortypdarstellung mit einer Zustandsspeicherung gearbeitet, wo die jeweiligen Funktionsabschlüsse nur einmal für die ganze Datenstruktur erzeugt werden. Da die z.Z. im System enthaltene Version durch ihren ausschließlich funktionalen Charakter die Effekte der implementierten Reduktions- und Expansionsalgorithmen des Optimierers besser veranschaulicht, werden die Beispiele in diesem Kapitel durch die vorgestellte funktionale *Iter*-Version formuliert. Die Realisierung der im nächsten Abschnitt skizzierten Algorithmen für die Seiteneffektanalyse, für die Code-Verschiebung und für die Eliminierung gemeinsamer Teilausdrücke würde effektive Optimierungen auch auf der imperativen *Iter*-Version zulassen.

```
Let Part = Tuple
  pNr :Int
  pName :String
  color :String
  weight :Int
  city :String
end

Let Supplier = Tuple
  sNr :Int
  sName :String
  status :Int
  city :String
end

Let SuppParts = Tuple
  sNr :Int
  pNr :Int
  quantity :Int
end
```

```
let parts = array
  tuple 1 "nut" "red" 12 "London" end
  tuple 2 "bolt" "green" 17 "Paris" end
  tuple 3 "screw" "blue" 17 "Roma" end
  tuple 4 "screw" "red" 14 "London" end
  tuple 5 "cam" "blue" 12 "Paris" end
  tuple 6 "cog" "red" 19 "London" end
end
let suppliers = array
  tuple 1 "Smith" 20 "London" end
  tuple 2 "Jones" 10 "Paris" end
  tuple 3 "Blake" 30 "Paris" end
  tuple 4 "Clark" 20 "London" end
  tuple 5 "Adams" 30 "Athens" end
end
```

```
let sp = array
  tuple 1 1 300 end
  tuple 1 2 200 end
  tuple 1 3 400 end
  tuple 1 4 200 end
  tuple 1 5 100 end
  tuple 1 6 100 end
  tuple 2 1 300 end
  tuple 2 2 400 end
  tuple 3 2 200 end
  tuple 4 2 200 end
  tuple 4 4 300 end
  tuple 4 5 400 end
end
```

Abbildung 6.3: Die *Part-Supplier*-Datenbank

**SQL:**

```
select name
from parts
where weight > 12
```

**TL:**

```
iter.get(fun(x :Part) x.pName
         arrayOp.elements(parts)
         fun(x :Part) x.weight > 12)
```

Die *get*-Funktion des Moduls *iter* stellt das funktionale Äquivalent einer Anfrage über einen Iterator dar. Sie bekommt als Argumente die Projektionsfunktion, den Iterator selbst, der durch die jeweilige *elements*-Funktion aus der Massendatenstruktur erzeugt wird, und das Selektionsprädikat. Damit entspricht sie dem gängigen *select-from-where*-Operator aus SQL.

Es ist offensichtlich, daß die funktionale Anfragenotation in TL im Gegensatz zu SQL dem ungeübten Benutzer manche Schwierigkeiten bereiten kann. Auf der anderen Seite bietet sie die Flexibilität, verschiedene Massendatentypen durch benutzerdefinierte Funktionen einheitlich zu verarbeiten. Diese zwei auseinandergehenden Eigenschaften – Deklarativität von SQL und Generalität von TL – können durch Syntaxerweiterungen vereint werden

[Schröder 93]. Ein solches Beispiel stellt eine auf TL aufgebaute, auf *Comprehensions* basierte Anfragesprache in [Juhacz 94] dar.

Auf jeden Fall ist die für den Optimierer relevante Form die TML-Darstellung, auf die alle aufgestockten Schichten letztendlich abgebildet werden. Aus diesem Grund sind die folgenden Effekte von der Existenz einer syntaktischen Erweiterung unabhängig.

## 6.3 Die Regeln der algebraischen Anfrageoptimierung

Nachdem auf die Analogien in der Architektur des allgemeinen Datenbankanfrageevaluators und des optimierenden TL-Übersetzers sowie auch auf die Eignung der universellen Programmiersprache TL für datenintensive Anwendungen hingewiesen wurde, ist die Frage interessant, inwieweit der TL-Optimierer die Anfrageevaluation effizienter gestalten kann. Die Anfrageoptimierung im Tycoon-System beruht auf den implementierten und in Abschnitt 5.2 beschriebenen allgemeinen codeverbessernden Transformationen.

Das Zusammenspiel der recht einfachen, aber doch generellen Grundregeln der Reduktions- und Expansionsphase (s. Abschnitt 5.3) mit der kompakten TML-Zwischencoderepräsentation und mit dem dynamischen Optimierungszeitpunkt (nach dem Binden) führt zu bestimmten optimierenden Codetransformationen bei der Anfrageevaluation innerhalb des persistenten Tycoon-Systems, ohne daß eine zielgerichtete Anfrageoptimierung vorgenommen wird. Selbstverständlich kann man nicht behaupten, daß diese Optimierungen einen „richtigen“ Anfrageoptimierer in einem solchen System ersetzen kann. Trotzdem schaffen sie eine Grundlage, welche die Probleme der Anfrageoptimierung auf der Implementationsebene von persistenten Systemen durchleuchtet und zu einem Versuch anregt, die Anfrageoptimierung auf die generelle Plattform der funktionalen Optimierung zu übertragen. Es existieren zwei Gründe, die einen solchen Ansatz rechtfertigen:

- In persistenten Systemen wird eine generelle Lösung des effizienten Zugriffs, Bearbeitung und Austausch von langlebigen Daten verlangt, da man im Extremfall jeden Zugriff auf den Objektspeicher als eine Anfrage betrachten kann.
- Ein reflektives System wie Tycoon mit seinem einheitlichen Bindungs-, Benennungs- und Typisierungskonzept und der Möglichkeit zur systeminternen Anbindung und Benutzung generischer Dienste verspricht einen globalen Überblick über alle in einem Codefragment involvierten Daten, System- und fremde Komponenten, der es dem dynamischen Optimierer gestatten kann, mehr Wissen für seine Transformationen zu verwenden, was zu einer erhöhten Laufzeiteffizienz führen würde.

Deswegen wäre eine interessante Aufgabe, Gemeinsamkeiten zwischen der allgemeinen Programmoptimierung und der bisher für die Datenbankwelt spezifischen Anfrageoptimierung zu erkennen und sie in persistenten Systemen zu realisieren. Die in der vorliegenden Arbeit diskutierten Optimierungstransformationen bieten eine Grundlage in diesem Sinne

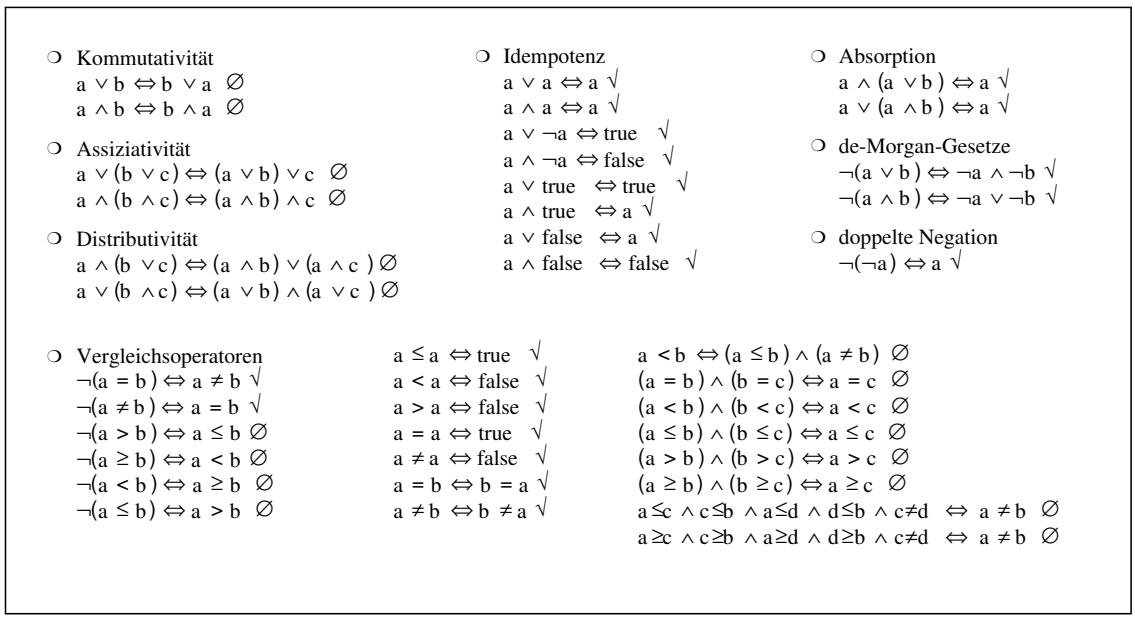



Abbildung 6.4: Evaluation boolescher Ausdrücke

und können als eine Vorbereitung auf kompliziertere und effektivere Optimierungsmethoden angesehen werden. In Abschnitt 5.4 wurden die Auswirkungen der Reduktions- und Expansionsregeln auf den TML-Code bestimmter TL-Sprachkonstrukte demonstriert. Im folgenden werden die mit den allgemeinen Optimierungstechniken (ohne den Einsatz der Datenflußanalyse) ausgewerteten und vereinfachten Elemente einer in TL über die *Iter*-Schnittstelle definierten Anfrage analysiert. Die dadurch realisierten Codevereinfachungen erfüllen einige der von der Datenbanktheorie bekannten algebraischen Transformationsregeln der funktionalen Anfrageoptimierung. Auf diese Fälle wird noch im Einzelnen eingegangen. Die funktionierenden Transformationen werden durch einige Beispiele präsentiert. Für weitere Optimierungen werden Ideen zu ihrer Realisierung skizziert.

Die größten Erfolge der TL-Optimierung können bei der Optimierung boolescher Prädikate registriert werden. Abb. 6.4 enthält eine Sammlung von algebraischen Transformationsregeln für boolesche Ausdrücke, die Teile von Anfrageselektionsprädikaten sein können. Die durch den TL-Optimierer funktionierenden Optimierungstransformationen werden durch $\surd$ gekennzeichnet, die nicht funktionierenden Transformationen durch $\oslash$. Man kann erkennen, daß die vom TL-Optimierer gelösten Prädikatoptimierungen (die Idempotenz-, Absorptions-, de-Morgan-Regeln, etc.) bestimmte Spezialfälle darstellen (z.B. Operandengleichheit oder konstanter Operand). In allen Fällen handelt es sich um eine Sequenz der Anwendung der allgemein im TL-Optimierer implementierten Regeln für *Inlining*, Beta-Reduktion, *if*-

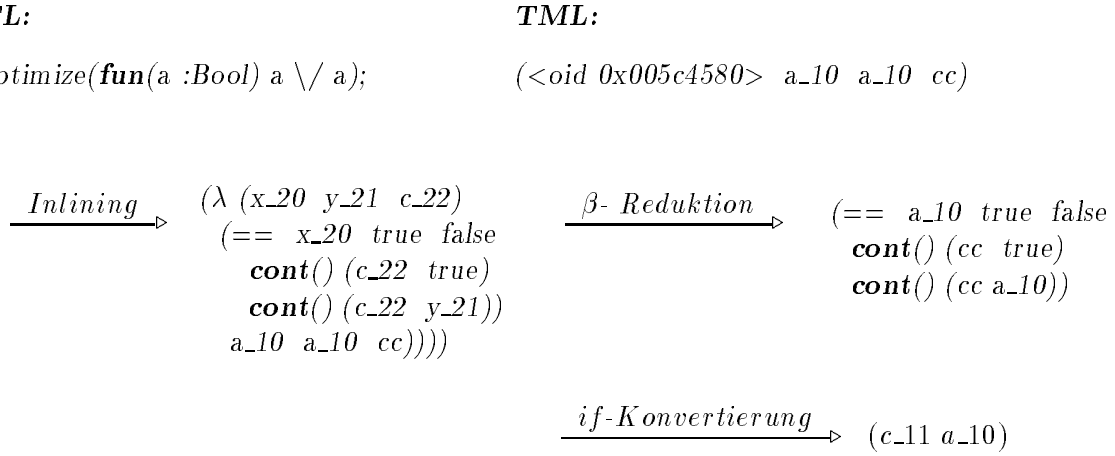



Abbildung 6.5: Anwendung der Idempotenzregel

Konvertierung und Auswertung konstanter Ausdrücke.

Um das Verfahren zu veranschaulichen, wird in Abb. 6.5 ein Beispiel für die Auswertung der linken Seite der ersten Idempotenzregel präsentiert. Man sieht, daß zunächst die *oder*-Funktion ('$\vee$') expandiert wird, was den Anlaß für eine Beta-Reduktion (Ersetzung der Funktionsparameter durch die aktuellen Argumente und Vereinfachung des eingesetzten Funktionsrumpfes) gibt. Die Anwendung der Grundregel *case-assign* erlaubt die Substitution von a_*10* im '=='-Zweig, der die frühe Evaluation der elementaren Operation '==' und die Rückgabe des berechneten Ergebnis folgt.

Bei anderen Regelgruppen, wie die Kommutativitäts-, Assoziativitäts-, Distributivitätsregeln, kann kein Erfolg der TL-Optimierung erwartet werden, da im TL-Übersetzer keine Interdependenzen und Äquivalenzen zwischen den verschiedenen elementaren TML-Operationen definiert sind. Außerdem bedarf der TL-Optimierer mancher komplizierterer Analyse, um globale Codeverschiebungen und Ersetzungen vorzunehmen. Diesen Bedarf skizziert folgendes Beispiel für die Evaluation der linken Seite der ersten Kommutativitätsregel:

***TL:***

*optimize(***fun***(x, y :Bool f, g :***Fun***(:Bool):Bool) f(x) \/ g(y));*

***TML:***

```
(f_17  x_15  cont(a_20)     Berechnung des ersten Operanden (a)
(g_18  y_16  cont(b_21)     Berechnung des zweiten Operanden (b)
(==  a_20  true  false      Implementation von '∨'
  cont() (cc  true)
  cont() (cc  b_21))))
```

Die Anwendung der Kommutativitätsregel in der Anfrageoptimierung beruht auf der eventuellen Erübrigung der Berechnung des zweiten (eventuell teureren) Operanden, falls der erste zu *true* evaluiert. Dies kann durch eine Hinauszögerung der Berechnung des zweiten (teureren) Operanden ermöglicht werden. Die allgemeine Äquivalenz der beiden Seiten dieser Regel kann folgendermaßen bei einer detaillierteren Optimierung ausgenutzt werden:

### Realisierung der Kommutatitivitätsregel

1. Kostenvergleich der Funktionen zur Operandenberechnung *f_17* und *g_18* nach folgender Heuristik:
   - falls nur eine der beiden Funktionen rekursive Funktionsaufrufe (Schleifen) in ihrem Rumpf enthält, dann wird sie als die teurere gekennzeichnet
   - falls dieses für beide gilt, werden keine Codeveränderungen vorgenommen
   - falls keine der beiden Funktionen rekursive Aufrufe (Schleifen) enthält, wird nach der Codegröße der Rümpfe entschieden
2. Vertauschen der beiden Resultatsvariablen a*_20* und *b_21*, so daß die mit der billigeren Berechnungsfunktion erzeugte Variable als '=='-Operand eingesetzt wird
3. Verschiebung der teureren Berechnung (*f_17* oder *g_18*) in den jeweiligen '=='-Zweig hinein, wenn diese keine Seiteneffekte enthält

Während Schritt 2 unproblematisch ohne Berücksichtigung von Seiteneffekten durchgeführt werden kann, erfordert die Verschiebung im letzten Schritt eine vorausgegangene Seiteneffektanalyse. In [Gifford, Lucassen 86] wird ein Ansatz zur Seiteneffektanalyse von Sprachen diskutiert, welche funktionale und imperative Eigenschaften aufweisen (wie TL). Dabei werden Funktionen vier hierarchisch angeordneten Seiteneffektkategorien zugewiesen: PURE (keine Seiteneffekte auf veränderbare Objekte), FUNCTION (Allokation von veränderbaren Objekten), OBSERVER (Erzeugung und lesende Operationen auf veränderbaren Objekten) und PROCEDURE (Erzeugung, Lesen und Modifikation veränderbarer Objekten). In diesem Zusammenhang werden auf der Funktionsebene zulässige Transformationen (z.B. Code-Verschiebung) definiert. Diese grobe Granularität bei der Seiteneffektanalyse hat jedoch den Nachteil, daß der Optimierer zu konservativ bleibt. Aus diesem Grund bietet sich eher folgende Vorgehensweise zur Seiteneffektanalyse im konkreten TML-Kontext, die variablenbezogen ist:

**Seiteneffektanalyse**

1. Bildung von 2 Mengen:
   - *def*-Menge, die alle im untersuchten Objekt neu allozierten veränderbaren Speicherpositionen mit den zugehörigen aktuellen Werten enthält und eine Multimenge darstellt
   - *use*-Menge, die alle veränderbaren Speicherpositionen enthält, die im untersuchten Objekt verwendet werden
2. Jede elementare Operation soll mit einer Funktion versehen werden, die diese zwei Mengen für konkrete Argumente generiert
3. Jeder Lambda-Knoten ist mit Feldern zur Speicherung der für seinen Rumpf berechneten Mengen ausgestattet
4. Bottom-Up-Berechnung der Seiteneffekte für die benutzerdefinierten Funktionen (zunächst für die Argumente, inclusive der Fortsetzungen, und dann für den Applikationsknoten selbst, d.h. dem Kontrollfluß entgegen):
   - die Umgebung bei der Baumtraversierung enthält die globalen *def*- und *use*-Mengen, die Bottom-Up erweitert werden
   - an jedem Applikationsknoten wird eine Vereinigung aus den globalen und konkreten (von der Funktionsposition bzw. den Argumenten stammenden) Mengen gebildet
   - an jedem Lambda-Knoten werden die bisher berechneten globalen Mengen in den jeweiligen Feldern notiert

Nachdem so alle Seiteneffekte für alle benutzerdefinierten Funktionen festgelegt worden sind, kann ein Applikationsknoten nach unten bzw. nach oben verschoben werden, solange sich seine (variablenbezogenen) Seiteneffekte nicht mit diesen an den übersprungenen Knoten überlappen (was technisch durch Durchschnittsbildung der jeweiligen *def*-Mengen zu realisieren ist). Bei Verzweigungen sollen entsprechende Maßnahmen getroffen werden, damit eine Code-Verschiebung nicht zu einer Veränderung der Codefunktionalität führen kann. Hierzu wird auf eine in [Appel 92] beschriebene allgemeine Strategie zur Baumtraversierung zwecks Code-Verschiebung (*hoisting*) hingewiesen.

Das skizzierte Verfahren zur Bestimmung der Seiteneffekte von benutzerdefinierten Funktionen kann weiteren globalen Optimierungstransformationen als Grundlage dienen. Insbesondere stellt es die Voraussetzung für die Eliminierung gemeinsamer Teilausdrücke (*common subexpression elimination*) dar. Als Veranschaulichung kann die Anwendung der Idempotenzregel auf beliebigen booleschen Ausdrücken dienen:

| ***TL:*** | ***TML:*** |
|---|---|
| *optimize(**fun**() f(a) \/ f(a));* | *(f_21 a_33 **cont**(t_14)*<br>*(f_21 a_33 **cont**(t_15)*<br>*(== t_14 true false*<br>***cont**() (cc true)*<br>***cont**() (cc t_15))* |

Es ist anzumerken, daß der Operand $a$ in der Idempotenzregel aus Abb. 6.4 ein beliebiger boolescher Ausdruck sein kann, während sich die Transformationen in Abb. 6.5 nur auf Variablen beziehen. Um die Idempotenzregel auf beliebigen Ausdrücken in TML zu implementieren, ist der Auswertung der elementaren Operation '==' eine Eliminierung gemeinsamer Teilausdrücke vorzuschalten. Dadurch kann die Gleichheit der Ergebnisse beider Ausdrücke (*t_14* und *t_15*) festgestellt und die Transformation aus Abb. 6.5 auf den gleichen Lambda-Variablen durchgeführt werden. Dazu sollte man den wiederholten Aufruf von $f$ mit demselben Argument ($a$) vermeiden, indem man die zweite Resultatsvariable *t_15* durch die erste (*t_14*) ersetzt. I.a. wird ein Verfahren zur Eliminierung gemeinsamer Teilausdrücke folgendermaßen aussehen:

**Eliminierung gemeinsamer Teilausdrücke**

1. Übergang zum nächsten Applikationsknoten (Top-Down-Traversierung des TML-Baumes); wenn dieser keine Seiteneffekte enthält, weiter mit 2, sonst mit 3
2. Suche nach dem nächsten Auftreten desselben Applikationsknotens im Code und Ausführung folgender Schritte:
   a) Entfernung des zweiten Applikationsknotens und Auswahl der jeweiligen Fortsetzung, die dem Fortsetzungszweig des ersten Applikationsknotens entspricht
   b) Substitution der Resultatsvariable in der Fortsetzung des zweiten Knotens durch die Resultatsvariable des ersten
   c) Wiederholung von Schritt 2 bis keine Duplikate des jeweiligen Applikationsknotens im TML-Code übrigbleiben
3. Wiederholung der Schritte 1 und 2, bis der ganze TML-Baum traversiert ist

Schritt 2 des obigen Algorithmus ist in Abb. 6.6 veranschaulicht.

Die hier beschriebenen Algorithmen, die im TL-Optimierer noch nicht implementiert sind, würden eine Reihe von weiteren Transformationen erlauben. Vor allem werden dadurch die imperativen TL-Elemente analysiert und optimiert. Auf der anderen Seite werden durch die Seiteneffektanalyse und Eliminierung gemeinsamer Teilausdrücke wiederholte Berechnungen vermieden, die besonders bei aktiverem Datenaustausch mit dem Objektspeicher

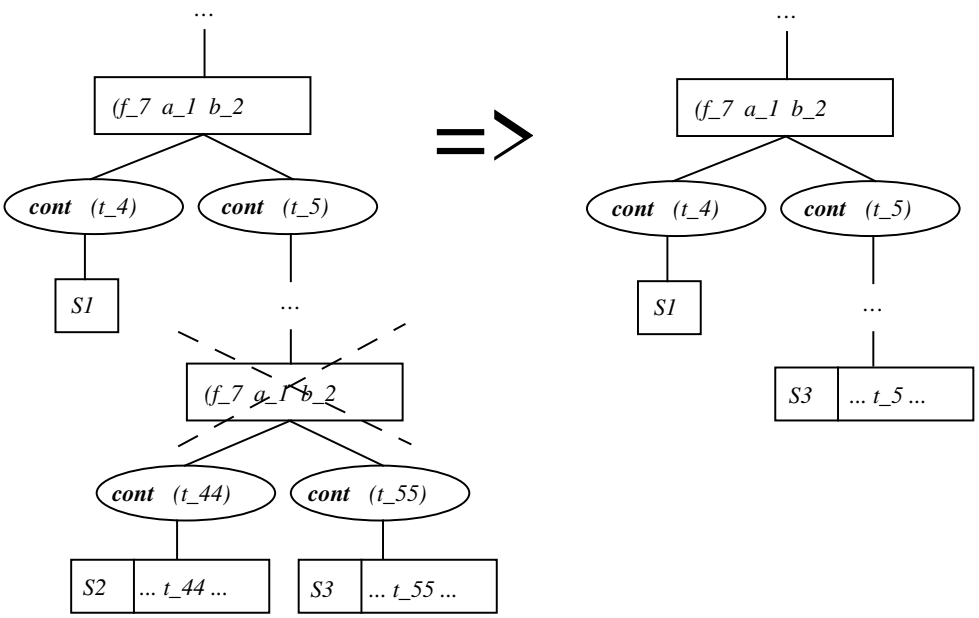



Abbildung 6.6: Eliminierung gemeinsamer Teilausdrücke

ins Gewicht fallen können. Und nicht zuletzt würden diese Algorithmen erlauben, die restlichen Transformationsregeln für die booleschen Ausdrücke in Anfrageprädikaten auf der TML-Seite zu realisieren.

Anhang C enthält eine ausführliche Sammlung von Transformationsregeln der relationalen Algebra, die in der funktionalen Anfrageoptimierung eingesetzt werden. Anders als bei der Prädikatevaluation bleiben die Auswirkungen der allgemeinen Optimierung lediglich auf der Ebene der allgemeinen Funktionsoptimierung. Die Implementation der verschiedenen Kommutativitäts-, Assoziativitäts- und Distributivitätsregeln sowie auch der Regeln zum Vertauschen selektiver (Projektionen und Selektionen) und konstruktiver (*Joins*, Kartesischer Produkte) Operationen erfordern eine kompliziertere Kostenanalyse in TML, sowie auch konkretes Wissen über den Charakter der einzelnen *iter*-Funktionen und über den aktuellen Datenbestand.

Eine technische Durchführung der notwendigen globalen TML-Transformationen könnte durchaus durch die Implementierung der Algorithmen für Seiteneffektanalyse, Code-Verschiebung (*hoisting*) und Eliminierung gemeinsamer Teilausdrücke ermöglicht werden. Es ist jedoch noch ein offenes Problem, inwieweit die aufwendigen semantischen Analysen und Anfragetransformationen in der TML-Repräsentation durchführbar sind. Eine Alternative dazu wären Optimierungen auf einer höher gelegenen Ebene, welche die Abhängigkeiten zwischen den Anfrageelementen besser erkennen läßt, wie z.B. die *comprehensions*-basierte

***TL:***

```
let pred12(x :Part) = x.weight > 12
let query = fun() iter.some(arrayOp.elements(parts) pred12)
optimize(query);
```

***Funktionsabschluß für 'query':***

```
proc(c_22)
  (λ(iter_18  arrayOp_19  parts_20  pred12_21)
    ([]  iter_18  33  cont(t_23)                          Zugriff auf 'some'
    ([]  arrayOp_19  12  cont(t_24)                       Zugriff auf 'elements'
    (t_24  parts_20  cont(t_25)
    (t_23  t_25  pred12_21  cont(t_26)
    (c_22  t_26)))))
    <oid 0x01137dec>                                      der Modul 'iter'
    <oid 0x012548b0>                                      der Modul 'arrayOp'
    <oid 0x013406c0>                                      das Feld 'parts'
    <oid 0x01646c04>)                                     die Funktion 'pred12'
```

***nach der dynamischen Optimierung:***

```
proc(c_22)
  (< oid 0x01254820 >  < oid 0x013406c0 >  1  7 cont(t_25)
   'arrayOp.elements'   'parts'
  (< oid 0x01137a28 >  t_25  < oid 0x01646c04 >  c_22))
   'iter.some'         Iterator  'pred12'
```

Abbildung 6.7: Optimierung von *iter.some*

Anfragesprache in [Juhacz 94]. Der Beitrag, den der TL-Optimierer bei der Anfrageevaluation momentan leistet, bleibt mehr auf der unteren Implementationsebene, vor allem in der geschachtelten Optimierung der von der Anfrage benutzten Funktionen und in der Prädikatevaluation.

Abb. 6.7 enthält ein Beispiel, an dem die bei der dynamischen Optimierung vorgenommenen Transformationen geschildert sind (vgl. Datenbankschema in Abb. 6.3 auf Seite 95). Auf der höchsten Ebene der dynamischen Optimierung (die *query*-Funktion selbst) werden die Funktionen *elements* und *some* aus den jeweiligen Moduln *arrayOp* bzw. *iter* herausgeholt und die entsprechenden Zugriffe eliminiert. Außerdem werden bei der geschachtelten dynamischen Optimierung die Funktionen *elements*, *some* und das Prädikat *pred12* in der Tiefe optimiert. Zu einer Expansion der *some*-Funktion kommt es hier nicht, da ihre Kosten über die *Savings* überwiegen (s. die Entscheidungsregel in Abb. 5.1 auf Seite 71). Der Grund dafür ist, daß die Iteratorstruktur durch eine Variable (*t_25*) repräsentiert ist, die das Ergebnis der Konvertierung des Felds *parts* durch *arrayOp.elements* aufnimmt. Also können die *Savings* des jeweiligen *some*-Parameters nicht berücksichtigt werden, da er nicht an ein konstantes Argument gebunden wird.

Um die Situation im Falle eines *Inlining* von *some* darzustellen, wird dieselbe Anfrage in Abb. 6.8 etwas anders formuliert. Der einzige Unterschied zu der Anfrage in Abb. 6.7 ist die explizite Bindung des durch *elements* erzeugte Iterator an eine TL-Variable, was die Allokation einer unveränderlichen Speicherstruktur für den Iteratorwert zur Folge hat. Nach dem Binden erkennt der TL-Optimierer die Unveränderlichkeit dieser mit einem *oid* referenzierten Speicherstruktur und kann daraus Elemente extrahieren (die Funktionen *empty*, *get* und *next* des Typs *Iter.T*). Dadurch erhöhen sich die *Savings* beim *some*-Aufruf und der Optimierer fällt die Entscheidung für die Expansion. Um denselben Effekt auch bei der ersten für den Benutzer gängigeren Anfrageformulierung zu erzielen, sollte eine Datenflußanalyse zur Bestimmung der erreichbaren Definitionen (s. [Aho et al. 86]) eingesetzt werden. Dadurch kann der Optimierer feststellen, daß *elements* ein unveränderliches Objekt zurückliefert, und dessen *Savings* bei der *Inlining*-Entscheidung berücksichtigen.

Aus dem letzten Beispiel kann man eindeutig erkennen, daß die allgemeinen Optimierungstechniken selbst bei der Anfrageevaluation zu Vereinfachungen führen können. Durch das *Inlining* der *some*-Funktion kann der Optimierer in die Massendatenstruktur herabsteigen und sie analysieren. Es läßt sich leicht erkennen, daß dadurch die Transformationen für leere Relationen bzw. für konstantes Prädikat (s. Anhang C) realisiert werden. Z.B. wird die *some*-Anfrage bei einem Prädikat, das zu *true* evaluiert vollständig ausgewertet:

| ***TL:*** | ***TML:*** |
|---|---|
| ***let*** *i = arrayOp.elements(parts);* | ***proc****(c_18)* |
| ***let*** *query =* ***fun****() iter.some(i* ***fun****(x :Part) true)* | *(c_18 true)* |
| *optimize(query);* | |

***TL:***

```
let pred12(x :Part) = x.weight > 12
let i = arrayOp.elements(parts);
let query = fun() iter.some(i pred12)
optimize(query);
```

***Funktionsabschluß für 'query':***

```
proc(c_21)
 (λ(iter_18  pred12_19  i_20)
  ([]  iter_18  33  cont(t_22)
  (t_22  i_20  pred12_19  cont(t_23)
  (c_21  t_23)))
  <oid 0x010e0ee4>                    der Modul 'iter'
  <oid 0x013703f4>                    die Funktion 'pred12'
  <oid 0x01370454>)                   der Iterator
```

***nach der dynamischen Optimierung:***

```
proc(c_21)
 (<oid 0x01370424>  cont(t_65)        Funktionskomponente 'get' von Iter.T
 ([]  t_65  4  cont(t_71)             Zugriff auf das 'weight'-Feld des Tupels
 (>  t_71  12                         die expandierte Prädikatfunktion
  cont()
   (c_21  true)
  cont()
   (<oid 0x01370408>  cont(t_68)      das Funktionsfeld 'next' von Iter.T
   (<oid 0x010e0b20>                  rekursive Anwendung von 'some'
      t_68  <oid 0x013703f4> c_21)))))  auf das nächste Feldelement
```

Abbildung 6.8: Optimierung von *iter.some* (mit vorangehender Bindung des Iterators)

Zusammenfassend kann man folgende Schlußfolgerungen für die Rolle der dynamischen Optimierung in Bezug auf die Anfrageoptimierung im persistenten Tycoon-System ziehen:

- Die z.Z. im TL-Übersetzer implementierten Reduktions- und Expansionstechniken lassen die Realisierung einer Reihe von Optimierungstransformationen verschiedener boolescher Ausdrücke als Teil von Anfrageprädikaten zu. Diese Transformationen sind ein Ergebnis des Zusammenspiels der Regeln für Funktionsexpansion, Beta-Reduktion, *if*-Konvertierung (die *case-assign*-Grundregel) und Auswertung konstanter Ausdrücke an den elementaren Operationen (die *fold*-Grundregel).

- Die Durchführung von Transformationen der relationalen Algebra, die Spezialfälle behandeln (wie leere Relationen bzw. konstante Prädikate) hängt im großen Maße von der Expansion der jeweiligen Anfragefunktion aus dem *iter*-Modul ab (*some*, *all*, *select*, *get*, etc.). Diese Transformationen lassen sich (bei funktionaler Iteratorimplementation) auf TML-Ebene mit allgemeinen Optimierungsmethoden realisieren.

- Einen besonders großen Beitrag für die realisierten Anfragetransformationen leistet der dynamische Charakter der TL-Optimierung, bei der alle von der Anfrage verwendeten Ressourcen systemweit dem Optimierer zur Verfügung stehen. Dadurch ist es überhaupt möglich, Anfrageoptimierung, deren dynamischer Charakter sich in der interpretativen Anfrageevaluation in Datenbanksystemen (s. Abschnitt 6.1) niederschlägt, zu betreiben.

- Um die reflexiven Transformationen (Kommutativität, Assoziativität, Distributivität, etc.) für die booleschen, Quantor- und relationalen Ausdrücke auf TML-Ebene implementieren zu können, ist einerseits eine Realisierung der Seiteneffektanalyse und solcher globaler Transformationen wie Code-Verschiebung und Eliminierung gemeinsamer Teilausdrücke notwendig. Anregungen für diese Elemente der Datenflußanalyse sind in [Aho et al. 86], [Fisher, LeBlanc 88], [Appel 92], [Dhamdhere et al. 92], [Knoop et al. 92], [Tjiang, Hennessy 92] und [Drechsler, Stadel 93] zu finden. Andererseits muß eine aufwendige Kostenanalyse betrieben werden, um die Richtung der Regelanwendung zu bestimmen.

- Man sollte einen vernünftigen Kompromiß zwischen der Generalität der in TML stattfindenden Optimierungstransformationen und dem damit verbundenen Analyseaufwand wegen des Informationsverlustes aus den höhersprachlichen Ebenen finden. In diesem Zusammenhang bietet sich die *Comprehensions*-Schnittstelle zur Realisierung einiger spezifischer Regeln der algebraischen Anfrageoptimierung (s. [Juhacz 94]) an.

# 7 Messungen und Beurteilung

Nachdem die Transformationen bei der dynamischen Optimierung von TL-Sprachkonstrukten (Abschnitt 5.4.1) und bei der Anfrageevaluation im Tycoon-System (Abschnitt 6.3) demonstriert wurden, wird in diesem Kapitel durch konkrete Messungen die Leistung des TL-Optimierers analysiert.

Die Messungen wurden auf zwei Gruppen von Programmen durchgeführt. Zum einen wurden die Standardtestprogramme aus der *Stanford Suite* und der *Richards' Benchmark*[1], die oft in der Literatur als Testgrundlage dienen, in TL implementiert. Zum anderen wurde der effizienzsteigernde Effekt der dynamischen Optimierungen auf großen TL-Programmpaketen wie dem TL-Übersetzer und dem TL-Parsergenerator analysiert. Als Hardware-Umgebung wurde ein *Sun SPARCserver 1000* benutzt, wobei die Zeit der Testdurchführung so gewählt wurde, daß die Maschine möglichst unbelastet ist, um die Ergebnisse nicht zu verzerren. Die Messungen wurden mit Hilfe der Funktion *times* aus der C-Standardbibliothek des Betriebssystems *Solaris 2.2* durchgeführt, welche die von einem Prozess verbrauchte CPU-Zeit zurückgibt [Solaris].

Im folgenden werden die Resultate präsentiert und bewertet. Abschnitt 7.1 stellt eine Gegenüberstellung der Meßwerte bei unoptimiertem, statisch optimiertem und dynamisch optimiertem TL-Code der Standardtestprogramme dar. Die Ergebnisse für die Performanzsteigerung werden in Abschnitt 7.2 auch anhand großer Programme wie des TL-Übersetzers und des TL-Parsers bestätigt. Dort wird auch der Optimierungsaufwand und die Codegrößeveränderungen berücksichtigt.

Während bei den erwähnten Messungen die vom Benutzer einstellbaren *Inlining*-Faktoren (*optimisticFactor* und *pessimisticFactor*, s. Abschnitt 5.3.2.1) auf aus der Erfahrung geeignete Werte gesetzt wurden, wird ihr Einfluß auf das *Towers*-Programm der Stanford Suite in Abschnitt 7.3 analysiert. Die dabei erzielten Ergebnisse können als Begründung der im Tycoon-System voreingestellten Erfahrungswerte dienen.

[1] Die C-Quelltexte für die *Stanford Suite* und der C++-Code für den *Richards' Benchmark* sind über dem Internet aus dem Verzeichnis /pub/benchmarks auf dem Rechner otis.stanford.edu (36.22.0.201) übernommen worden (vgl. [Chambers 92]).

| Programm | nicht optimiert (s) | statisch optimiert (s) | statisch relativ (%) | dynamisch optimiert (s) | dynamisch relativ (%) |
|---|---|---|---|---|---|
| *Bubble* | 53 s | 51 s | 96 % | 34 s | 64 % |
| *Perm* | 14 s | 14 s | 100 % | 8 s | 57 % |
| *FFT* | 68 s | 59 s | 87 % | 54 s | 79 % |
| *Mm* | 28 s | 26 s | 93 % | 21 s | 75 % |
| *Tree* | 15 s | 13 s | 87 % | 9 s | 60 % |
| *Quick* | 29 s | 27 s | 93 % | 18 s | 62 % |
| *Intmm* | 22 s | 19 s | 86 % | 17 s | 77 % |
| *Towers* | 23 s | 21 s | 91 % | 14 s | 60 % |
| *Queens* | 43 s | 34 s | 79 % | 29 s | 67 % |
| *Puzzle* | 93 s | 92 s | 99 % | 79 s | 85 % |
| *Richards'* | 233 s | 227 s | 97 % | 136 s | 58 % |
| Durchschnitt | | | 92 % | | 68 % |

Tabelle 7.1: Performanzmessungen an den Standardtestprogrammen der *Stanford Suite* und am *Richards' Benchmark* – TL-Implementation

## 7.1 Messungen an Standardtestprogrammen

Die *Stanford Suite* enthält 10 Testbeispiele, die sich aus der Erfahrung insbesondere zu Performanzmessungen von Sprachen der Modula-Familie gut eignen. Sie verwenden verschiedenartige Sprachkonstrukte wie rekursive Funktionen (besonders *Perm*, *Towers*, *Queens*, *Tree*), Schleifen und Feldbearbeitungen (vor allem in *Puzzle* und *FFT*). Obwohl sie mehr imperativ orientiert sind, wurden sie in die Tests des TL-Optimierers übernommen, um einen Bezug zu den in der Literatur enthaltenen Testergebnisse zu ermöglichen. Der *Richards' Benchmark* ist ein etwas größeres Programm, in dem das Szenario eines Betriebssystems simuliert wird.

Tab. 7.1 enthält die Werte für die Ausführungszeiten der 11 in TL implementierten Standardtestprogramme in der nicht optimierten, statisch optimierten und dynamisch optimierten Version. Die Tabellenwerte stellen die vom jeweiligen Programm beanspruchte Prozessorzeit dar. Es ist offensichtlich, daß die statische Optimierung einen geringen Beitrag zur Effizienzerhöhung leistet. Der schon erwähnte hohe Grad an Modularität des Tycoon-

| Programm | nicht optimiert (s) | optimiert (s) | relativ (%) | TL/C | TL/C (optimiert) |
|---|---|---|---|---|---|
| *Bubble* | 0.07 s | 0.02 s | 29 % | 757 | 1700 |
| *Perm* | 0.05 s | 0.01 s | 20 % | 280 | 800 |
| *FFT* | 0.12 s | 0.03 s | 25 % | 567 | 1800 |
| *Mm* | 0.05 s | 0.03 s | 60 % | 560 | 700 |
| *Tree* | 0.39 s | 0.30 s | 77 % | 38 | 30 |
| *Quick* | 0.05 s | 0.02 s | 40 % | 580 | 900 |
| *Intmm* | 0.10 s | 0.06 s | 60 % | 220 | 283 |
| *Towers* | 0.07 s | 0.02 s | 29 % | 329 | 700 |
| *Queens* | 0.04 s | 0.02 s | 50 % | 1075 | 1450 |
| *Puzzle* | 0.36 s | 0.10 s | 28 % | 258 | 790 |
| *Richards'* | 0.99 s | 0.21 s | 21 % | 235 | 648 |
| Durchschnitt | | | 40 % | 445 | 891 |

Tabelle 7.2: Performanzmessungen an den Standardtestprogrammen der *Stanford Suite* und am *Richards' Benchmark* – C-Implementation

Systems schränkt die Möglichkeiten des Optimierers, Codetransformationen vorzunehmen, nur innerhalb eines Moduls ein. Da selbst die elementaren arithmetischen Operationen auch aus fremden Moduln beim Binden importiert werden müssen, kann der Optimierer während der Übersetzung sogar an solchen Stellen keine Vereinfachungen vornehmen.

Um so mehr wird die Bedeutung der Wahl des dynamischen Zeitpunkts für die Optimierung (nach dem Binden) deutlich. Hier wird der Code im Durchschnitt in 68 % der Zeit ausgeführt, die bei den jeweiligen nicht optimierten Programmversionen benötigt wird. Diese deutliche Verbesserung der Laufzeiteffizienz ist auf den hohen Informationsstand zurückzuführen, den der TL-Optimierer nach dem Binden besitzt. Dadurch wird die Besonderheit des Tycoon-Systems deutlich, welche die erheblich deutlicheren Effizienzvorteile der dynamischen Optimierung im Vergleich zu [Srivastava, Wall 92] erklärt. Die eindeutige Schlußfolgerung aus diesen Daten ist, daß sich für extrem modulare Systeme wie Tycoon ausschließlich eine dynamische Optimierung empfehlen läßt.

Aus Tab. 7.1 kann man außerdem erkennen, daß die Resultate bei den verschiedenen Programmen schwanken. Das ist durch die in den einzelnen Programmen jeweils dominierenden Sprachkonstrukte bedingt. Programme, die vorwiegend funktional aufgebaut sind (z.B.

| Versionen | nicht optimiert | | -O1 | | -O2 | | -O3 | |
|---|---|---|---|---|---|---|---|---|
| | absolut | relativ | absolut | relativ | absolut | relativ | absolut | relativ |
| lokal | 0.238 s | 100 % | 0.205 s | 86 % | 0.201 s | 84 % | 0.098 s | 41 % |
| modular | 0.245 s | 100 % | 0.201 s | 82 % | 0.194 s | 79 % | 0.143 s | 58 % |

Tabelle 7.3: Einfluß der Modularität auf den GNU-C-Übersetzer am Beispiel von „Türme von Hanoi“

eher rekursive Funktionen als Schleifen und Zustandsvariablen verwenden), weisen eine annähernd doppelte Geschwindigkeit der dynamisch optimierten Version auf (*Perm*, *Tree*, *Towers*), während andere, die auf mehreren modifizierenden Feldoperationen und Schleifen basieren (wie *Puzzle* und *FFT*) eine geringere Laufzeitverbesserung zeigen. Auf der einen Seite ist diese Tatsache mit der allgemein komplizierteren Optimierung von modifizierbaren Werten zu erklären. Auf der anderen Seite eignen sich die implementierten Reduktions- und Expansionstransformationen vor allem für einen funktionalen Programmierstil.

Eine Erweiterung des dynamischen TL-Optimierers mit Elementen der Datenflußanalyse, für die einige Algorithmen in Abschnitt 6.3 präsentiert worden sind, wird vermutlich zu einer deutlicheren Verbesserung auch solcher Programmteile führen, die imperativ aufgebaut sind. Als gute Kandidaten sind hier die Code-Verschiebung (*hoisting*) sowie die Eliminierung gemeinsamer Teilausdrücke zu nennen, die z.B. das Herausziehen teuerer, konstanter Berechnungen aus zyklischen Codeteilen bzw. die Vermeidung wiederholter Berechnungen ermöglichen würden.

Es ist nicht gerade fair das persistente Tycoon-System in seiner Performanz mit anderen nicht persistenten Systemen zu vergleichen. In vielen Literaturquellen über optimierende Übersetzer ([Kranz et al. 86], [Pendergrast, Ryder 86], [Chambers 92]) gilt der C-Übersetzer auf den jeweiligen Rechnerarchitekturen als Vergleichsbasis (mit der Option *-O2*). Die Bemühung ist, seine allgemein anerkannte hohe Optimierungsqualität auch bei höhersprachlichen Umgebungen (z.B. LISP [Kranz et al. 86] bzw. objekt-orientierten Systemen [Chambers 92]) zu erreichen, die weitere vorteilhafte Eigenschaften besitzen. Um diese Tradition fortzuführen, bietet Tab. 7.2 einen Vergleich der in TL implementierten (persistenten) Programme zu ihren (flüchtigen) C-Äquivalenten an.

Es ist anzumerken, daß in der vorliegenden Arbeit im Unterschied zu [Chambers 92] die volle Leistung (inclusive *Inlining*) des GNU-C-Optimierers auf dem *Sun SPARCserver 1000* getestet wurde (Übersetzung der C-Programme mit der Option *-O3*, der C++-Programme mit der Option *-O4*). An der durchschnittlichen Performanzsteigerung des TL- und C-Codes er-

| relevante Größen | C unopt./opt. | TL unopt./opt. | Ingres Windows4GL | VisualBASIC ohne DB/mit DB |
|---|---|---|---|---|
| Hardware | Sun SPARCserver1000 | | Sun SPARCserver470 | 80486/33 kHz/16 MB |
| Laufzeit | 0.238 s/0.143 s | 16 s/8 s | 349 s | 11 s/5250 s |
| Normier.-koeffizient | 1 | 1 | 0.65 | 0.32 |
| normierte Zeit | 0.238 s/0.143 s | 16 s/8 s | 227 s | 3.5 s/1680 s |

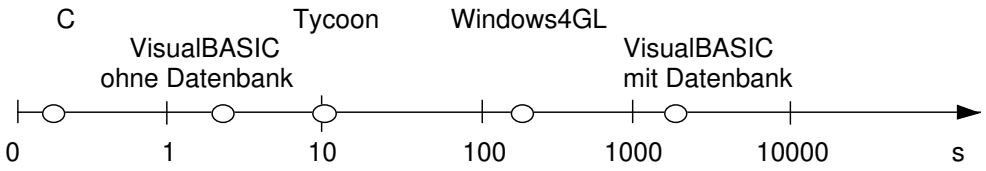



Abbildung 7.1: Performanzvergleich zwischen C, Tycoon, Ingres Windows4GL und VisualBASIC (am Beispiel von „Türme von Hanoi“)

kennt man, daß der TL-Optimierer etwas schlechter als der C-Optimierer abschneidet. Man kann jedoch erwarten, daß man in Tycoon an die Qualität der C-Optimierung herankommen wird, wenn auch die veränderbaren Werte effizient optimiert werden.

Die volle Leistung des C-Optimierers kann jedoch erreicht werden, weil alle Testprogramme lokal (innerhalb eines Moduls) implementiert sind. Während die Modularität keine Auswirkung auf die dynamische TL-Optimierung hat, sieht es anders bei der (statischen) C-Optimierung aus. Tab. 7.3 stellt die lokale und modulare (aus 3 Moduln bestehende) Version von „*Türme von Hanoi*“[2] gegenüber. Bei Optimierungen auf dem Objektcode (Übersetzung mit *-O1*) bzw. lediglich codereduzierenden Transformationen (*-O2*) wird keinen Unterschied zwischen der lokalen und modularen Version beobachtet. Bei einer Optimierung mit der Option *-O3*, die auch automatisch *Inlining* durchführt, ist ein klarer Unterschied zu Lasten der

[2]Im Unterschied zu *Towers* aus der *Stanford Suite* ist diese Version durch drei Stapel mit Speicherallokationen implementiert.

modularen Version zu verzeichnen. Das Laufzeitverhältnis der modularen C-Version von „*Türme von Hanoi*“ (58 %) ist sogar schlechter als dieses für die TL-Version[3] (50 %, s. Abb. 7.1). Es ist offensichtlich, daß bei modularen Anwendungen der mächtige aber statische C-Optimierer schlechter als der auf Codereduktion und -expansion beruhende dynamische TL-Optimierer abschneidet.

Die vielen hervorragenden Eigenschaften des Tycoon-Systems wie Persistenz, Portabilität, Reflektion, dynamische Bindung, Abstraktion, etc. werden durch diesen gewaltigen Unterschied zur Performanz der Sprache C erkauft. Z.B. darf man nicht vergessen, daß das Aufsetzen auf einem Objektspeicher eine bedeutende Auswirkung auf das Laufzeitverhalten des Codes hat, da die Zugriffszeit auf ein sekundäres Speichermedium im Durchschnitt $10^3$ mal größer ist als die auf ein primäres. Außerdem muß man berücksichtigen, daß der TVM-Code interpretiert wurde. In [Mathiske 92] ist die Möglichkeit zur Generierung von C-Code aus der TML-Repräsentation beschrieben. Gemäß der im zitierten Bericht enthaltenen Ergebnisse bringt die Ausführung des generierten C-Codes unter Ausnutzung der C-Optimierungen einen 50-fachen Effizienzvorteil.

Abb. 7.1 bietet einen Performanzvergleich von Tycoon und zwei weiteren Systemen, Ingres Windows4GL [Ingres 90] und VisualBASIC [Maslo, Dittrich 93], die mit ihrer Persistenz und ihrem interpretativen Charakter konzeptionell nah an Tycoon sind. Die Messungen wurden anhand von *Türme von Hanoi* (einem der Programme aus der *Stanford Suite*) durchgeführt. Wegen der Verfügbarkeit der Systeme auf unterschiedlichen Hardware-Plattformen wurden Normierungskoeffizienten auf der Basis der C-Version des Programms berechnet. Die ermittelten normierten Zeiten deuten auf die gute Performanz des Tycoon-Systems im Vergleich zu anderen persistenten Programmierumgebungen mit interpretativen Charakter.

## 7.2 Messungen an großen TL-Programmen

Obwohl die Testprogramme in Abschnitt 7.1 eine breite Verwendung finden, können sie nicht in vollem Maße die Eigenschaften des Tycoon-Systems ausschöpfen, da sie vor allem aus der Welt der imperativen Sprachen stammen, die keine reichhaltigen Konstrukte wie Polymorphie, Funktionen höherer Ordnung und Subtypisierung besitzen.

Um eine Antwort auf die Frage zu geben, wie sich die Optimierung bei voller Ausnutzung der Tycoon-Eigenschaften auswirkt, wurden zwei große TL-Programmpakete – der TL-Übersetzer selbst und der TL-Parsergenerator – als Testobjekte verwendet. Darauf wurden nicht nur die Laufzeiteffizienz, sondern auch die Effekte der Optimierung in ihrer Auswirkung auf Codegröße und Übersetzungszeit analysiert. Beide Testobjekte eignen sich gut für realitätsnahe Aussagen über die Eigenschaften der TL-Optimierung, da sie zwei umfassende Tycoon-Systempakete darstellen, die von verschiedenen Personen geschrieben worden sind (d.h., daß sie verschiedene Programmierstile und Sprachkonstrukte aufweisen) und von den

[3] Anhang D bietet den TL-Code, sowie den unoptimierten und dynamisch optimierten TML-Code des Beispiels „*Türme von Hanoi*“ an.

| Programm | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|
| TL-Übersetzer (198 Moduln) | 23333 s | 23792 s | 16276 s | 22139 s | 17654 s |
| relativ zu V1 | | | 70 % | 95 % | 71 % |
| TL-Parsergenerator | 133 s | 134 s | 103 s | 123 s | 101 s |
| relativ zu V1 | | | 77 % | 92 % | 76 % |

Tabelle 7.4: Laufzeit und relative Laufzeiteffizienz der TL-Übersetzer- und TL-Parsergeneratorversionen

modernen höhersprachlichen TL-Konstrukten profitieren.

Es wurden insgesamt fünf Übersetzerversionen erzeugt und anschließend deren Eigenschaften bei der vollständigen Neuübersetzung des TL-Übersetzers selbst und bei der Generierung des TL-Parsers (mit LL(1)-Grammatik) gemessen. Die gemessenen Übersetzer- bzw. Parsergeneratorversionen sind im Einzelnen:

**V1:** die nicht optimierte Version, ohne PTML-Codegenerierung

**V2:** die nicht optimierte Version, mit PTML-Codegenerierung

**V3:** die dynamisch optimierte Version (mit PTML-Codegenerierung)

**V4:** die statisch optimierte Version (mit PTML-Codegenerierung)

**V5:** die sowohl statisch als auch dynamisch optimierte Version (mit PTML-Codegenerierung)

In Tab. 7.4 sind die einzelnen Messungen angeboten. Die erste Zeile dieser Tabelle enthält die jeweilige Übersetzungszeit des gesamten TL-Übersetzers (als Objekt der Messungen) mit den verschiedenen Testversionen des Übersetzers (als Subjekte der Messungen). Die zweite Zeile bezieht sich auf die 5 Versionen der Generatorfunktion, mit deren Hilfe der TL-Parser erzeugt wird. Die Werte darin bezeichnen die Laufzeit der jeweiligen Parsergenerierung. Die Tabelle bestätigt die Aussagen aus dem vorangegangenen Abschnitt. Man sieht, daß der dynamisch optimierte TL-Übersetzer nur 70 % der Zeit des nicht optimierten benötigt, um sich selbst neu zu generieren. Das entspricht einer 43 %-gen Erhöhung der Übersetzungsgeschwindigkeit. Die statisch optimierte Übersetzerversion jedoch bringt keinen nennenswerten Effizienzvorteil.

| Übersetzerphasen | ohne PTML-Generierung | | mit PTML-Generierung | |
|---|---|---|---|---|
| | absolut | relativ | absolut | relativ |
| Statische Übersetzung | 14662 s | 100 % | 15811 s | 100 % |
| Parsen | 1294 s | 9 % | 1346 s | 9 % |
| Typüberprüfung | 9892 s | 67 % | 10087 s | 64 % |
| TL-nach-TML-Transformation | 474 s | 3 % | 467 s | 3 % |
| Statische Optimierung | 1378 s | 9 % | 1391 s | 9 % |
| Statische Codegenerierung | 1186 s | 8 % | 2073 s | 13 % |
| Dynamische Optimierung | | | 1811 s | 11 % |
| Dynamische Codegenerierung | | | 1114 s | 7 % |
| Dynamische Übersetzung | | | 2925 s | 18 % |

Tabelle 7.5: Übersetzerphasen (mit und ohne PTML-Codegenerierung)

Es ist interessant zu beobachten, daß die Version V5 (statische und dynamische Optimierung) nicht besser als die Version V3 (nur dynamische Optimierung) abschneidet. Diese Tatsache wird durch die Unfähigkeit der statischen Optimierung erklärt, einen bedeutsamen Beitrag zur Performanz modularer Anwendungen zu leisten, so daß die dynamische Optimierung ausschlaggebend bleibt.

Eine interessante Fragestellung der Optimierung ist es, wie hoch der (relative) Optimierungsaufwand ist und wie er sich zu den anderen Übersetzungsphasen verhält. Die Antwort darauf ist in Tab. 7.5 enthalten, wo die Ergebnisse für die Neuerzeugung des TL-Übersetzers mit (V2) und ohne PTML-Codegenerierung (V1) erfaßt sind. Als Bezugswert ist die Summe aller Zeiten für die separate Übersetzung der übersetzerrelevanten Moduln gewählt. Die Tabelle besagt, daß sich der Optimierungsaufwand sowohl in der statischen als auch in der dynamischen Phase in Grenzen hält. Trotz der wiederholten Umstrukturierungen des TML-Baumes beträgt der Anteil der statischen Optimierung nur 9 % der statischen Übersetzung. Nicht viel größer (11 %) fällt die noch leistungsfähigere dynamische Optimierung aus, so daß für die dynamisch optimierte Version ein zusätzlicher Aufwand von 18 % gegenüber der statischen Übersetzungszeit entsteht. Dazu zählt auch die Zeit für die Neugenerierung und für das Anlegen der geschachtelt optimierten Funktionsobjekte im Objektspeicher.

| Programm | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|
| TL-Übersetzer<br>relativ zu V1 | 529 kB | 1165 kB<br>220 % | 871 kB<br>165 % | 1119 kB<br>212 % | 929 kB<br>176 % |
| TL-Parsergenerator<br>relativ zu V1 | 186 kB | 310 kB<br>167 % | 248 kB<br>133 % | 305 kB<br>164 % | 295 kB<br>159 % |

Tabelle 7.6: Codegröße der TL-Übersetzer- und TL-Parsergeneratorversionen

Wenn man die Codegenerierungszeit der beiden Übersetzerversionen betrachtet, stellt man fest, daß die PTML-Generierung den Zeitaufwand dieser Übersetzungskomponente beinahe verdoppelt. Dieser negative Effekt der dynamischen Optimierung (für deren Zwecke die zusätzliche Codedarstellung PTML neben dem ausführbaren TVM-Code erzeugt wird) läßt sich vermutlich durch Verschmelzung der beiden linearen Repräsentationen (TVM und PTML) vermeiden. Ein weiteres Ziel bei der Entwicklung des *Back-End* des TL-Übersetzers ist diese Codevereinheitlichung.

Ein weiterer negativer Effekt der doppelten Codegenerierung ist aus Tab. 7.6 zu entnehmen: der ausführbare Programmcode nimmt nach einer PTML-Codegenerierung mehr als den doppelten Speicherplatz ein. Auf der anderen Seite sieht man, daß sich die Optimierung nicht nur performanzsteigernd, sondern auch speicherplatzreduzierend auswirkt. Der Vergleich der Codegrößen- und Laufzeitverhältnisse (Tab. 7.4 und Tab. 7.6) liefert einen Hinweis auf die Eignung der in der *Inlining*-Entscheidungsregel (Abb. 5.1 auf Seite 71) benutzten Heuristik für die Kosten-*Savings*- bzw. Codegröße-Laufzeitverhalten-Analyse. Anhand der Codegrößereduktion bilden die Programmversionen folgende Rangordnung: den höchsten Beitrag leistet die dynamische Optimierung, dann die statisch-dynamische, und am wenigsten sind die Auswirkungen der statischen Optimierung.

## 7.3 Optimierungsparameter

In Abschnitt 5.3.2.1 wurden bereits zwei vom Benutzer einstellbare Faktoren des *Inlining* erwähnt: der optimistische und der pessimistische Faktor. Mit diesen Faktoren kann der Benutzer ähnlich wie in [Appel 92] die Aggressivität der Expansion und damit auch der ganzen Optimierung beeinflußen.

Mit dem optimistischen Faktor kann der Optimierungsaufwand zugunsten einer größeren

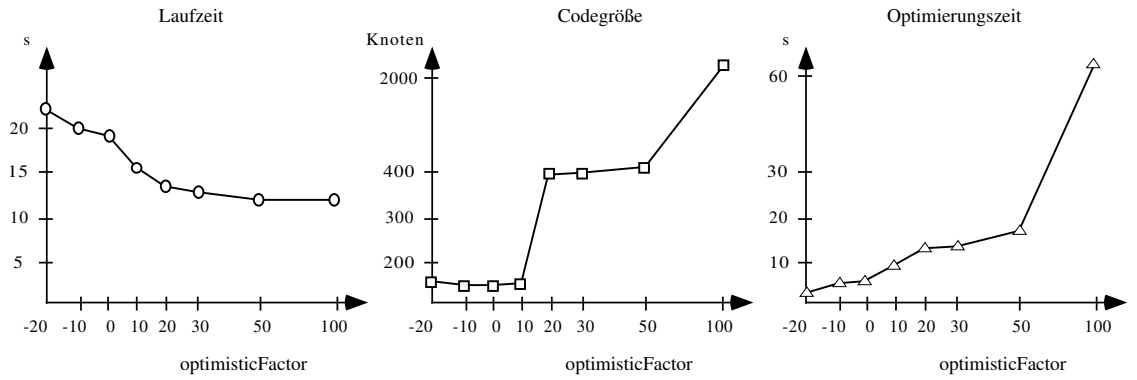



Abbildung 7.2: Einfluß von *optimisticFactor* am Beispiel von *Towers*

Freiheit bei der Funktionsexpansion durch Regulierung der Kostenschwelle (getrennt für die rekursiven und nicht rekursiven Funktionen) in der Entscheidungsregel von Abb. 5.1 (S. 71) erhöht bzw. gesenkt werden. Bei den bisherigen Messungen wurden sie auf Erfahrungswerte eingestellt, die im Durchschnitt das beste Performanzergebnis liefern. In Abb. 7.2 ist der Einfluß des optimistischen Faktors auf die drei wesentlichen Größen der Optimierung anhand des *Towers*-Beispiels der *Stanford Suite* grafisch dargestellt. Der optimistische Faktor[4] wurde im Bereich zwischen $-20$ und $+100$ verändert und dabei Laufzeit, (dynamische) Optimierungszeit und Codegröße (als TML-Knotenzahl) gemessen. Es lassen sich leicht folgende Anhängigkeiten erkennen:

**Laufzeit:** Mit der Erhöhung des optimistischen Faktors reduziert sich die Laufzeit bis zu einem Sättigungswert, nach dem jede weitere Erhöhung des Faktors keinen weiteren Einfluß auf die Performanz hat.

**Codegröße:** Bei der die Codegröße repräsentierenden TML-Knotenzahl beobachtet man eine steigende Tendenz in Abhängigkeit vom optimistischen Faktor. Im Bereich zwischen $-20$ und $+20$ kann man kaum eine Veränderung feststellen, während bei einer Erhöhung des Faktors auf einen Wert insbesondere über $50$ ein gewaltiger Aufstieg festzustellen ist. Der letzte ist auf die zu liberale Expansion zurückzuführen, nach der jedoch in der Regel keine weitere Reduktion zustandekommt. Ursache für diese schlechte Schätzung der künftigen Codereduktionen ist die Minderung des Einflusses der *Savings* in der Entscheidungsregel durch die zu hohen konstanten Faktoren (s. Abb.5.1 auf Seite 71).

[4] Wie es schon in Abschnitt 5.3.2.1 erwähnt wurde, kann der optimistische *Inlining*-Faktor getrennt für rekursive und nicht rekursive Funktionen eingestellt werden. Bei den Messungen wurde jedoch der Einfachheit halber keinen Unterschied zwischen den verschiedenen Funktionen gemacht.

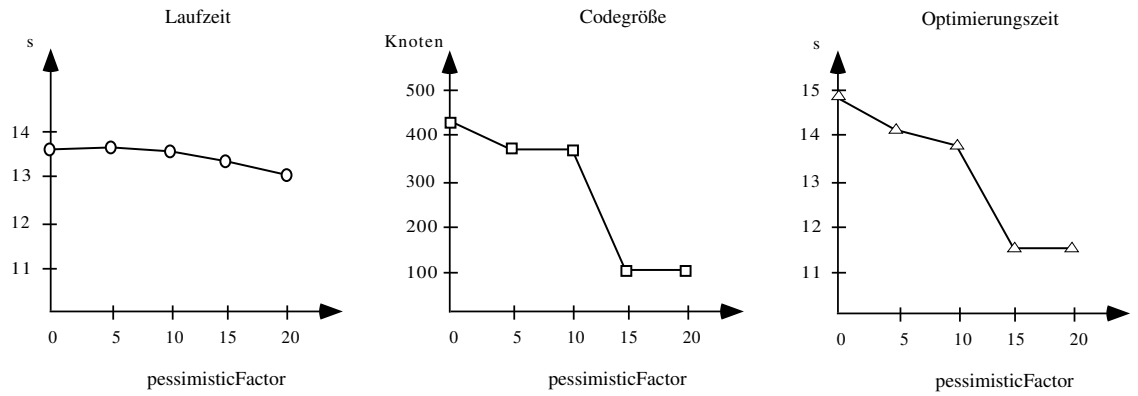



Abbildung 7.3: Einfluß von *pessimisticFactor* am Beispiel von *Towers*

**Optimierungszeit**: Es ist naheliegend, daß die Optimierungszeit zusammen mit der Codegröße bei Erhöhung des optimistischen *Inlining*-Faktors steigt, wobei diese Steigung unbegrenzt ist (z.B. durch *Inlining* rekursiver Funktionen).

Abb. 7.3 gibt die Wirkung des negativen *Inlining*-Faktors (*pessimisticFactor*) auf die Optimierung an. Dieser Faktor wirkt sich auf die Terminierungsgeschwindigkeit der Optimierung aus, indem er mit steigender Anzahl der Optimierungsläufe die zulässige Kostenschwelle der zu expandierenden Funktionen progressiv senkt. Er wird im Intervall zwischen *0* und *20* verändert, wobei negative Werte sinnlos sind, da sie einen entgegengesetzten Effekt bei der Terminierung aufweisen würden. Da jedoch dieser Faktor ab dem zweiten Optimierungslauf seine Wirkung zeigt und im ersten Lauf eine Reihe von *Inlinings* passieren können, bleibt die Performanz immer besser als diese des unoptimierten Codes.[5]

Die Minderung der Codegröße und Optimierungszeit ist zu erwarten, da mit immer steigendem pessimistischem *Inlining*-Faktor Codeveränderungen nur im ersten Optimiererzyklus (auf allen Schachtelungsebenen der dynamischen Optimierung) vorgenommen werden. Es ist interessant, daß beim Nullwert des pessimistischen Faktors kein extrem großes Maximum weder bei der Codegröße noch bei der Optimierungszeit zu registrieren ist. Der Grund dazu ist, daß die Terminierung des Algorithmus in diesem Fall indirekt durch die erhöhten Kosten der expandierten Funktionen in den nachfolgenden Optimiererzyklen erzwungen wird. Bei einer Erhöhung von *pessimisticFactor* über *15* pendeln sich die TML-Knotenzahl und die Optimierungszeit auf einen durch die im ersten Lauf stattfindenden Optimierungstransformationen festgelegten Wert ein.

[5]Im jeweiligen Programm ist die Auswirkung der Änderung von *pessimisticFactor* jedoch unbedeutend, da alle möglichen Expansionen bereits im ersten Optimiererzyklus getätigt werden. Dies wäre nicht der Fall bei anderen Programmen, wo wesentliche Transformationen nach dem ersten Zyklus vorgenommen werden.

Abb. 7.3 demonstriert eindeutig die Rolle des pessimistischen *Inlining*-Faktors: er reguliert die Anzahl der Optimiererzyklen (jedoch nach dem ersten) und bestimmt dadurch lediglich die Terminierungsgeschwindigkeit, hat jedoch einen kleineren Einfluß auf die Laufzeiteffizienz (indem er sich auf die Transformationen nach dem ersten Optimiererzyklus auswirkt).

Das Zusammenspiel der insgesamt drei (2 optimistischen für rekursive und nicht rekursive Funktionen und eines pessimistischen) *Inlining*-Faktoren verändert das allgemeine Verhalten des TL-Optimierers: ob er eher konservativ ist, zugunsten eines kleineren Optimierungsaufwands, oder ob er eher aggressiv zugunsten einer besseren Laufzeitperformanz ist. Der Benutzer kann diese Faktoren von der Programmierumgebung aus individuell einstellen. Im Tycoon-System sind jedoch Erfahrungswerte für die Faktoren voreingestellt (mit denen auch die Messungen in Abschnitt 7.1 und Abschnitt 7.2 durchgeführt wurden):

$optimisticFactor = 25$
$optimisticFactorRec = 15$
$pessimisticFactor = 5$

Ein Vergleich mit den Abbildungen 7.2 und 7.3 zeigt, daß auch für dieses konkrete Beispiel die Erfahrungswerte im optimalen Bereich sowohl für die Laufzeiteffizienz als auch für Optimierungsaufwand und Codegröße liegen.

# 8 Zusammenfassung und Ausblick

Das persistente und hoch modulare Tycoon-System mit seiner polymorphen, strikt typisierten Programmiersprache TL bietet eine bequeme Programmierumgebung zur Integration generischer Dienste in einem einheitlichen Benennungs-, Bindungs- und Typisierungsschema. Die innovativen Tycoon-Konzepte stellen eine Herausforderung für die Optimierung, insbesondere zur Steigerung der Laufzeitperformanz, dar.

Wie in traditionellen optimierenden Übersetzern durchläuft der TL-Code bei der Übersetzung mehrere Repräsentationen, von denen die TML-Repräsentation als Grundlage für optimierende Transformationen gewählt wird. In der vorliegenden Arbeit wird die besondere Eignung dieser CPS-basierten Zwischensprache für die optimierungsrelevanten Codeanalysen und -restrukturierungen bestätigt.

Es wird ein Optimierungsmodell erarbeitet, formal spezifiziert und implementiert. Ausgehend von einer kleinen Anzahl von Grundregeln werden eine Reihe von Optimierungstechniken wie Beta-Reduktion, *Inlining*, Auswertung konstanter Ausdrücke, *if*-Konvertierung, Kopien-Propagierung, etc. realisiert.

Als einen entscheidenden Faktor für die Performanzsteigerung hat sich die dynamische Wahl des Zeitpunktes der Optimierung erwiesen. Aufgrund des hohen Modularitätsgrades des gesamten Tycoon-Systems verliert die statische Optimierung (während der separaten Übersetzung) an Bedeutung, während sich die dynamische Optimierung nach einer Rücktransformation des gebundenen und im Objektspeicher abgelegten Codes als besonders wirksam erweist. Sowohl die demonstrierten Auswirkungen auf verschiedene TL-Sprachkonstrukte als auch die an Standardtestprogrammen und an großen TL-Programmpaketen gemessenen Ergebnisse bestätigen diese These. So werden Performanzsteigerungen bis zu 50 % durch die dynamische Optimierung beobachtet.

Durch geeignete Heuristiken, spezielle Verfahren zur Codetraversierung und durch die Ausnutzung der vom System angebotenen impliziten Persistenz wird der TL-Optimierer selbst auch unter besonderer Beachtung von Effizienzgesichtpunkten entworfen. Trotz der separaten Aufrufbarkeit des dynamischen Optimierers bleibt der Aufwand bei der Generierung von effizientem Code in vertretbaren Grenzen (ca. 18 % gegenüber der reinen Übersetzung). Zusätzlich besteht die Möglichkeit, die Schwerpunktsetzung im Kompromiß zwischen Laufzeitperformanz und Optimierungsaufwand durch Parameter zu regulieren.

Besondere Aufmerksamkeit wird der Nutzung der universellen Programmiersprache TL als Datenbanksprache gewidmet. Dabei werden die Auswirkungen der dynamischen Optimierung auf durch das Tycoon-Werkzeug zur Iterationsabstraktion formulierten Anfragen analysiert. Neben der Auswertung von Anfrageprädikaten und der Realisierung einiger algebraischer Anfragetransformationen gibt der aktuelle Stand des TL-Optimierers Aufschluß über weitere Chancen zur Übertragung der Probleme der algebraischen Anfrageoptimierung auf die allgemeine Implementationsebene der generellen Zwischencodeanalyse und -transformationen in persistenten Systemen.

Aufgrund der hier vorgelegten Untersuchungen der Anfrageoptimierung und der durchgeführten Messungen bieten sich folgende Möglichkeiten zur Weiterentwicklung des TL-Optimierers bzw. des TL-Übersetzers an:

- Verschmelzung der ausführbaren TVM-Codedarstellung und des generierten PTML-Codes unter Beibehaltung der Vorteile der beiden linearen Coderepräsentationen: Effizienz bei der Ausführung bzw. Abbildungsfähigkeit und Wiedererkennbarkeit höhersprachlicher Konstrukte (insbesondere rekursiver Funktionen).
- Realisierung einer Seiteneffektanalyse zur Behandlung modifizierbarer Werte im TML-Code. Darauf basierend lassen sich verschiedene Datenflußanalysen durchführen, die über ein generisches Werkzeug angeboten werden könnten.
- Eine Implementierung von Algorithmen für Code-Verschiebung (*hoisting*) und Eliminierung gemeinsamer Teilausdrücke würde die Fähigkeit des TL-Optimierers verbessern, Schleifen, veränderbare Werte und Felder zu optimieren. Außerdem können sie als Grundlage für die Realisierung einer Reihe von reflexiven algebraischen Transformationen für Datenbankanfragen unter Einbeziehung einer geeigneten Kostenbewertung dienen.

Diese Erweiterungen lassen (wenn auch mit einer Erhöhung des Optimierungsaufwands) eine weitere Steigerung der Laufzeiteffizienz erwarten, so daß die Implementation persistenter Programme auch unter dem Performanzaspekt immer realistischer wird. In seiner derzeitigen Form hat der TL-Optimierer mit der Anwendung von *Inlining* und Reduktionen über Modulgrenzen hinweg vermutlich den größten relativen Beitrag für die Effizienzsteigerung des generierten Codes bereits geleistet und eine Grundlage für weitere Untersuchungen und Ausbau geschaffen.

# A TML Datenstruktur

```
Let Rec EvalFn = All(:T)T      (* Typ der Evaluationsfunktion *)
and CostFn = All(:T)Int        (* Typ der Kostenfunktion *)

(* TML-Baumknoten *)

and T = Tuple
  case primitive with          (* elementare Operation *)
    name :String               (* benutzerdefinierter Name *)
    info :PrimitiveInfo        (* Optimiererinformation *)
    eval :EvalFn               (* Evaluationsfunktion *)
    cost :CostFn               (* Kostenfunktion *)
    nValueArgs :Int            (* Anzahl der Wertargumente *)
    nContArgs :Int             (* Anzahl der Fortsetzungsargumente *)
    nResults :Int              (* Anzahl der zurückgegebenen Resultate *)
  end
  (* Elementare Operationen können nur an Funktionspositionen erscheinen *)

  case variable with
    name :String
    fCont :Bool                (* 'true' für Fortsetzungsvariablen *)
    subst :T                   (* an die Variable gebundener Substitutionswert *)
    nReadRefs :Int             (* Anzahl der Variablenreferenzen *)
    nApplyRefs :Int            (* Anzahl der Variablenreferenzen
                                  an Funktionsposition *)
    isGlobal :Bool             (* 'true' für globale Variablen *)
    savings :Int               (* potentielle Savings der Variable *)
  end

  case literal with
    value :Literal
  end
```

```
  case readRef with              (* Variablenreferenz *)
    variable :T
  end

  case lambda with
    variables :list.T(T)         (* Parameterliste *)
    body :T                      (* muß ein apply-Knoten sein *)
    isRec :Bool                  (* 'true' für rekursive Abstraktionen *)
    unknown :Int                 (* Anzahl der globalen Referenzen *)
    cost :Int                    (* Kosten der Abstraktion *)
  end

  case apply with
    fn :T                        (* primitive-, readRef-, literal- oder lambda-Knoten *)
    args :list.T(T)              (* Argumentenliste *)
  end
end (* T *)

and Literal = Tuple
  case nil with
  case literals with value :Array(Literal)
  case tml with value :T
  case bool with value :Bool
  case char with value :Char
  case int with value :Int
  case real with value :Real
  case string with value :String
  case arrayLit with value :Array(T)
  case oid with
    value :tm.OID                (* Objektidentität *)
    store :tm.Store              (* Objektspeicheridentität *)
    tml :T                       (* aus PTML konvertierte Lambda-Abstraktion
                                    (nur für Funktionen, sonst leer) *)
end
```

# B Spezifikation der Optimierungsalgorithmen

## B.1 Notation

Die zur Beschreibung der folgenden Optimierungsalgorithmen verwendete Notation lehnt sich an [Plotkin 81], [Cardelli 90] und [Matthes 92a] an. Die Definitionen in diesem Kapitel bauen auf der in Abb. 4.1 auf Seite 31 präsentierten abstrakten Syntax von TML sowie auf der in Abschnitt 5.1 definierten Syntax für die Grundregeln der Optimierung in Tycoon auf.

### B.1.1 Grundlagen

| | |
|---|---|
| $tml \in TML$ | TML-Knoten |
| $\rho \in Env$ | Umgebung |
| $subst \in Subst \subseteq Env$ | Substitutionspaare |
| $val/v$ | Variable/Wert-Paar ($v$ ist an $val$ gebunden) |
| $val^e \in Val$ | Fortsetzungswerte |
| $val^t \in Val$ | direkte Werte (Werte erster Klasse) |
| $oid \in Oid \subset Lit$ | Objektidentifikatoren (Objektspeicherreferenzen) |
| $nRounds$ | Anzahl der Optimiererzyklen |
| $\|tml\|_v$ | Anzahl der Referenzen auf $v$ im Teilbaum $tml$ |
| $\|use(v)\|$ | Anzahl aller Referenzen auf $v$ |
| $nApplyRefs(v)$ | Anzahl der Referenzen auf $v$ an Funktionsposition |
| $changed(tml) \in Env$ | $true$, wenn der TML-Baum $tml$ durch irgendeine Regel modifiziert wurde |
| $isRec(val)$ | $true$, wenn der Wert eine rekursive Abstraktion ist |
| $isKnown(oid)$ | $true$, wenn das $oid$ bei der Traversierung bereits getroffen wurde |
| $isClosure(oid)$ | $true$, wenn das $oid$ ein Funktionsobjekt referenziert |
| $unknownRefs(abs)$ | Anzahl der globalen Referenzen in $abs$ |
| $traversedArgs(app)$ | $true$, wenn die Argumente von $app$ schon traversiert worden sind |

$$
\begin{array}{lll}
val & ::= & val^c \mid val^t \\
tml & ::= & val \mid app \\
Subst & ::= & \oslash \mid Subst, val/v
\end{array}
$$

### B.1.2 Regeln

$$KomplexeRegel ::= \frac{Prämissen}{Regel} \quad \Big(Bedingungen\Big)$$

$$Prämissen ::= \oslash \mid Regel \;\; Prämissen \mid \begin{array}{c} Prämissen \\ Prämissen \end{array}$$

### B.1.3 Abkürzungen

$$<val_n> \;::= val_1 \; val_2 \; \ldots \; val_n$$

$$<val/v_n> \;::= val_1/v_1 \; val_2/v_2 \; \ldots \; val_n/v_n$$

$$<val_n> \setminus val_i ::= val_1 \; \ldots \; val_{i-1} \; val_{i+1} \; \ldots \; val_n$$

$$<val_n> \;\xrightarrow[\rho]{R}\; <val', \rho'_n> \;::= \quad val_1 \xrightarrow[\rho]{R} val'_1, \rho'_1 \; \ldots \quad val_n \xrightarrow[\rho]{R} val'_n, \rho'_n$$

$$<val_n> \;\xrightarrow[<(\rho, v)_n>]{R}\; <(val', \rho')_n> \;::=$$

$$val_1 \xrightarrow[\rho_1,\; v_1]{R} val'_1, \rho'_1 \; \ldots \quad val_n \xrightarrow[\rho_n,\; v_n]{R} val'_n, \rho'_n$$

## B.2 Der Reduktionsalgorithmus

$$tml : TML \xrightarrow[\rho : Env]{\textbf{reduce}} tml : TML,\ \rho' : Env$$

$$\frac{app \xrightarrow[\rho]{reduce} app', \rho' \quad \lambda\ (<v_n>\ )\ app' \xrightarrow[\rho']{\eta\text{-}reduce} val, \rho''}{\lambda\ (<\ v_n\ >)\ app \xrightarrow[\rho]{reduce} val, \rho''}$$

$$\frac{v \xrightarrow[\rho]{subst} val, \rho' \quad val \xrightarrow[\rho', v]{substRec} val', \rho''}{v \xrightarrow[\rho]{reduce} val', \rho''}$$

$$\frac{<val_n> \xrightarrow[\rho]{reduce} <val'_n>\ ,\ \rho'}{(l\ <val_n>\ ) \xrightarrow[\rho]{reduce} (l\ <val'_n>\ ),\ \rho'}$$

$$<val_n> \xrightarrow[\rho]{trav\text{-}\lambda\text{-}appArgs} <val'_n>\ , \rho'$$

$$\frac{(\lambda\ (<v_n>\ )\ app\ <val'_n>\ ) \xrightarrow[\rho']{reduce\text{-}\lambda\text{-}apply} app', \rho''}{(\lambda\ (<\ v_n\ >)\ app\ \ <\ val_n\ >) \xrightarrow[\rho]{reduce} app', \rho''}$$

$$\frac{v \xrightarrow[\rho]{subst} val, \rho' \quad (val <val_n> ) \xrightarrow[\rho']{reduce} app, \rho''}{(v <val_n> ) \xrightarrow[\rho]{reduce} app, \rho''} \quad \Big(val/v \in \rho\Big)$$

$$\frac{<val_n> \xrightarrow[\rho]{reduce} <val'_n> , \rho'}{(v <val_n> ) \xrightarrow[\rho]{reduce} (v <val'_n> ), \rho'} \quad \Big(val/v \notin \rho\Big)$$

$$(Y\ \lambda\ (c <v_n> )(c\ \textbf{\textit{cont}}()app <abs_n> )) \xrightarrow[\rho]{bind} \rho'$$

$$<abs_n> \xrightarrow[\rho']{reduce} <abs'_n> , \rho'' \quad app \xrightarrow[\rho'']{reduce} app', \rho'''$$

$$(Y\ \lambda\ (c <v_n> )(c\ \textbf{\textit{cont}}()app' <abs'_n> )) \xrightarrow[\rho''']{Y\text{-}reduce} app'', \rho''''$$

$$app'' \xrightarrow[\rho'''']{tryFold} app''', \rho'''''$$

$$\frac{}{(Y\ \lambda\ (c <v_n> )(c\ \textbf{\textit{cont}}()app <abs_n> )) \xrightarrow[\rho]{reduce} app''', \rho'''''}$$

$$\frac{<val^t_n> \xrightarrow[\rho]{reduce} <val^{t'}{}_n> , \rho' \quad (p <val^{t'}{}_n> <val^c_m> ) \xrightarrow[\rho']{tryFold} app, \rho''}{(p <val^t_n> <val^c_m> ) \xrightarrow[\rho]{reduce} app, \rho''} \quad \Big(p \in Prim \setminus Y\Big)$$

$$tml : TML \xrightarrow[\rho : Env,\ v_{old} : Var]{\textbf{substRec}} tml : TML,\ \rho' : Env$$

$$\frac{v \xrightarrow[\rho]{reduce} val, \rho'}{v \xrightarrow[\rho, v_{old}]{substRec} val, \rho'} \quad \left(v \neq v_{old}\right)$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{traverse-}\lambda\textbf{-appArgs}} tml : TML,\ \rho' : Env$$

$$(\lambda\ (< v_n >)\ app\ \ < val_n >) \xrightarrow[\rho]{\eta\text{-}reduce} app', \rho'$$

$$\frac{}{(\lambda\ (< v_n >)\ app\ \ < val_n >) \xrightarrow[\rho]{trav\text{-}\lambda\text{-}appArgs} app', \rho', traversedArgs(app') = false}$$

$$\frac{v \xrightarrow[\rho]{subst} val, \rho' \quad val \xrightarrow[\rho']{trav\text{-}\lambda\text{-}appArgs} val'', \rho''}{v \xrightarrow[\rho]{trav\text{-}\lambda\text{-}appArgs} val'', \rho''}$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{reduce-}\lambda\textbf{-apply}} \triangleright\, tml : TML,\ \rho' : Env$$

$$(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{bind} \triangleright\, \rho' \quad app\ \xrightarrow[\rho']{reduce} \triangleright\, app', \rho''$$

$$(\lambda\ (<v_n>\ )\ app'\ <val_n>\ )\ \xrightarrow[\rho'']{eliminateArgs} \triangleright\, (\lambda\ (<v'_m>\ )\ app'\ <val'_m>\ ),\ \rho'''$$

$$<val'_m>\ \ \xrightarrow[\rho''']{reduce} \triangleright\, <val''_m>\ ,\ \rho''''$$

$$(\lambda\ (<v'_m>\ )\ app'\ <val''_m>\ )\ \xrightarrow[\rho'''']{reduce\text{-}\lambda\text{-}appRec} \triangleright\, app'', \rho'''''$$

$$\frac{}{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{reduce\text{-}\lambda\text{-}apply} \triangleright\, app'', \rho'''''}$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{reduce-}\lambda\textbf{-appRec}} \triangleright\, tml : TML,\ \rho' : Env$$

$$(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{removeAbstraction} \triangleright\, app', \rho'$$

$$\frac{}{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{reduce\text{-}\lambda\text{-}appRec} \triangleright\, app', \rho'}$$

$$\left(\forall val_i \in <val_n>\ : val_i \notin Lit \bigwedge\ val_i \notin Var\right)$$

$$(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{reduce\text{-}\lambda\text{-}apply} app', \rho'$$

$$app'\ \xrightarrow[\rho']{removeAbstraction} app'', \rho''$$

$$\frac{}{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{reduce\text{-}\lambda\text{-}appRec} app'', \rho''}$$

$$\left(\exists val_i \in <val_n>\ : val_i \in Lit\ \bigvee\ val_i \in Var\right)$$

$$tml : TML\ \xrightarrow[\rho : Env]{\textbf{tryFold}} tml : TML,\ \rho' : Env$$

$$(p\ <val_n>\ )\ \xrightarrow[\rho]{fold} app, \rho'\quad app\ \xrightarrow[\rho']{tryFoldRec} app', \rho''$$

$$\frac{}{(p\ <val_n>\ )\ \xrightarrow[\rho]{tryFold} app', \rho''}$$

$$tml : TML\ \xrightarrow[\rho : Env]{\textbf{tryFoldRec}} tml : TML,\ \rho' : Env$$

$$v\ \xrightarrow[\rho]{subst} val, \rho'\quad (val\ <val_n>\ )\ \xrightarrow[\rho']{tryFoldRec} app, \rho''$$

$$\frac{}{(v\ <val_n>\ )\ \xrightarrow[\rho]{tryFoldRec} app, \rho''}$$

$$\frac{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{reduce} app', \rho'}{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{tryFoldRec} app', \rho'}$$

$$(==\ val_0\ <val_n^t>\ \ <val_m^c>\ )\ \xrightarrow[\rho]{case\text{-}assign} <\rho'_m>$$

$$<val_m^c>\ \ \xrightarrow[<\rho'_m>]{reduce} <(val^{c'}, \rho'')_m>$$

$$\frac{(==\ val_0\ <val_n^t>\ \ <{val^{c'}}_m>\ )\ \xrightarrow[\rho,\ traversedArgs(app_0) = true]{tryFold} app, \rho'''}{(==\ val_0\ <val_n^t>\ \ <val_m^c>\ )\ \xrightarrow[\rho,\ traversedArgs(app_0) = false]{tryFoldRec} app, \rho'''}$$

$$\Big(app_0 = (==\ val_0\ <val_n^t>\ \ <val_m^c>\ )\Big)$$

$$<val_m^c>\ \ \xrightarrow[\rho]{reduce} <{val^{c'}}_m>\ ,\ \rho'$$

$$\frac{(p\ <val_n^t>\ \ <{val^{c'}}_m>\ )\ \xrightarrow[\rho',\ traversedArgs(app_0) = true]{tryFold} app, \rho''}{(p\ <val_n^t>\ \ <val_m^c>\ )\ \xrightarrow[app,\ \rho,\ traversedArgs(app_0) = false]{tryFoldRec} \rho''}$$

$$\Big(p \in Prim\ \setminus\ '==' \bigwedge\ app_0 = (p\ <val_n^t>\ \ <val_m^c>\ )\Big)$$

## B.3 Der Expansionsalgorithmus

$$tml : TML \xrightarrow[\rho : Env]{\textbf{expand}} tml : TML,\ \rho' : Env$$

$$\frac{app \xrightarrow[\rho]{expand} app', \rho'}{\lambda\ (< v_n >)\ app \xrightarrow[\rho]{expand} \lambda\ (< v_n >)\ app',\ \rho'}$$

$$\frac{l \xrightarrow[\rho]{unloadOid} val, \rho'}{l \xrightarrow[\rho]{expand} val, \rho'}$$

$$\frac{< val_n > \xrightarrow[\rho]{expand} < val'_n >\ ,\ \rho' \quad (v < val'_n >\ ) \xrightarrow[\rho']{inlineCall} app, \rho''}{(v < val_n >\ ) \xrightarrow[\rho]{expand} app, \rho''}$$

$$\frac{< val_n > \xrightarrow[\rho]{expand} < val'_n >\ ,\ \rho' \quad l \xrightarrow[\rho']{unloadOid} val_0, \rho'' \quad (val_0 < val'_n >\ ) \xrightarrow[\rho'']{inlineCall} app, \rho'''}{(l < val_n >\ ) \xrightarrow[\rho]{expand} app, \rho'''}$$

$$(Y\ \lambda\ (c\ <v_n>\ )(c\ \textbf{cont}()app\ <abs_n>\ ))\ \xrightarrow[\rho]{bind}\ \rho'$$

$$<v_n>\ \xrightarrow[\rho']{reverse}\ <v'_n>\ ,\ \rho' \quad <abs_n>\ \xrightarrow[\rho']{reverse}\ <abs'_n>\ ,\ \rho'$$

$$<abs'_n>\ \xrightarrow[\rho']{expand}\ <abs''_n>\ ,\ \rho'' \quad app\ \xrightarrow[\rho'']{expand}\ app', \rho'''$$

$$\frac{}{(Y\ \lambda\ (c\ <v_n>\ )(c\ \textbf{cont}()app\ <abs_n>\ ))\ \xrightarrow[\rho]{expand}\ app_0,\ \rho'''}$$

$$\Big(app_0 = (Y\ \lambda\ (c\ <v'_n>\ )(c\ \textbf{cont}()app'\ <abs''_n>\ ))\Big)$$

$$\frac{<val_n>\ \xrightarrow[\rho]{expand}\ <val'_n>\ ,\ \rho'}{(p\ <val_n>\ )\ \xrightarrow[\rho]{expand}\ (p\ <val'_n>\ ),\ \rho'} \quad \Big(p \in Prim \setminus Y\Big)$$

$$(\lambda\ (<\ v_n\ >)\ app\ <\ val_n\ >)\ \xrightarrow[\rho]{bind}\ \rho' \quad <val^t_n>\ \xrightarrow[\rho']{expand}\ <val^{t'}_n>\ ,\ \rho''$$

$$\lambda\ (<\ v_n\ >)\ app\ \xrightarrow[\rho'']{expand}\ abs, \rho''' \quad (abs\ <val^{t'}_n>\ )\ \xrightarrow[\rho''']{eliminateArgs}\ app', \rho''''$$

$$\frac{}{(\lambda\ (<\ v_n\ >)\ app\ <\ val_n\ >)\ \xrightarrow[\rho]{expand}\ app', \rho''''}$$

$$in : List(TML) \xrightarrow[\rho : Env]{\textbf{reverse}} \triangleright\, out : List(TML),\ \rho : Env$$

$$<val_n> \xrightarrow[\rho]{reverse} \triangleright\, val_n\ val_{n-1}\ \ldots\ val_1,\ \rho \qquad \Big(nRounds\ modulo\ 2 = 0\Big)$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{unloadOid}} \triangleright\, tml : TML,\ \rho' : Env$$

$$\frac{oid[i] \xrightarrow[\rho]{unloadOid} \triangleright\, oid'[i], \rho'}{oid \xrightarrow[\rho]{unloadOid} \triangleright\, oid', \rho'} \qquad \Big(isArray(oid) \bigwedge not(isKnown(oid))\Big)$$

$$\frac{abs \xrightarrow[\rho,\ abs/oid]{nestedOptimize} \triangleright\, val, \rho'}{oid \xrightarrow[\rho,\ abs/oid]{unloadOid} \triangleright\, val, \rho'}$$

$$\Big(isClosure(oid) \bigwedge not(isKnown(oid)) \bigwedge abs = ptml2tml(oid)\Big)$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{nestedOptimize}} tml : TML,\ \rho' : Env$$

$$\frac{abs \xrightarrow[\rho]{reduce} abs', \rho' \qquad \Big(unknownRefs(abs) = 0\Big)}{abs \xrightarrow[\rho]{nestedOptimize} abs', \rho'}$$

$$\frac{abs \xrightarrow[\rho,\ changed(abs) = true]{nestedOptimizeLoop} abs', \rho' \qquad \Big(unknownRefs(abs) \neq 0\Big)}{abs \xrightarrow[\rho]{nestedOptimize} abs', \rho'}$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{nestedOptimizeLoop}} tml : TML,\ \rho' : Env$$

$$\frac{abs \xrightarrow[\rho]{reduce} abs', \rho' \quad abs' \xrightarrow[\rho',\ changed(abs') = false]{expand} abs'', \rho'' \quad abs'' \xrightarrow[\rho'']{nestedOptimizeLoop} abs''', \rho'''}{abs \xrightarrow[\rho,\ changed(abs) = true]{nestedOptimizeLoop} abs''', \rho'''}$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{inlineCall}} tml : TML,\ \rho' : Env$$

$$\frac{l \xrightarrow[\rho]{inline} val, \rho' \quad val \xrightarrow[\rho']{nestedInline} val', \rho''}{l \xrightarrow[\rho]{inlineCall} val', \rho''}$$

$$\frac{v \xrightarrow[\rho]{inline} val, \rho' \quad val \xrightarrow[\rho']{nestedInline} val', \rho''}{v \xrightarrow[\rho]{inlineCall} val', \rho''} \quad \Big(isRec(v)\Big)$$

$$\frac{\begin{array}{c} v \xrightarrow[\rho]{subst} val, \rho' \quad val \xrightarrow[\rho']{unloadOid} val', \rho'' \quad val' \xrightarrow[\rho'']{inline} val'', \rho''' \\ val'' \xrightarrow[\rho''']{nestedInline} val''', \rho'''' \end{array}}{v \xrightarrow[\rho]{inlineCall} val''', \rho''''} \quad \Big(not(isRec(v))\Big)$$

$$tml : TML \xrightarrow[\rho : Env]{\textbf{nestedInline}} tml : TML,\ \rho' : Env$$

$$\frac{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{bind}\ \rho' \quad \lambda\ (< v_n >)\ app\ \xrightarrow[\rho']{expand}\ val,\ \rho'' \qquad (val\ < val_n >\ )\ \xrightarrow[\rho'']{eliminateArgs}\ app', \rho'''}{(\lambda\ (< v_n >)\ app\ \ < val_n >)\ \xrightarrow[\rho]{nestedInline}\ app', \rho'''}$$

$$\Big(\lambda\ (< v_n >)\ app \notin Cont \bigwedge\ \exists val_i \in < val_n >\ : (nApplyRefs(v_i) > 0 \bigwedge val_i \notin Var)\Big)$$

# C Transformationen der relationalen Algebra

Im folgenden ist eine Sammlung von Transformationsregeln der relationalen Algebra angeboten, auf deren Grundlage die Auswirkungen der TL-Optimierungen auf Datenbankanfragen im Tycoon-System untersucht worden sind. Diese Sammlung ist aus [Koch 85], [Dörfler, Peschek 88], [Ullman 88a], [Ullman 88b], [Klein 90], [Beeri, Kornatzki 91], [Subieta 91], [Abiteboul, Beeri 92] unter Verwendung einer einheitlichen Notation zusammengestellt worden.

## C.1 Quantorausdrücke

1. **Kommutativität**

   a) $\exists r_1 \in rel_1,\ \exists r_2 \in rel_2 : p(r_1, r_2) \iff \exists r_2 \in rel_2,\ \exists r_1 \in rel_1 : p(r_1, r_2)$
   $\left(rel_1 \notin scope(r_2) \ \bigwedge\ rel_2 \notin scope(r_1)\right)$

   b) $\forall r_1 \in rel_1,\ \forall r_2 \in rel_2 : p(r_1, r_2) \iff \forall r_2 \in rel_2,\ \forall r_1 \in rel_1 : p(r_1, r_2)$
   $\left(rel_1 \notin scope(r_2) \ \bigwedge\ rel_2 \notin scope(r_1)\right)$

   c) $\exists r_1 \in rel_1,\ \forall r_2 \in rel_2 : p(r_1, r_2) \iff \forall r_2 \in rel_2,\ \exists r_1 \in rel_1 : p(r_1, r_2)$
   $\left(rel_1 \notin scope(r_2) \ \bigwedge\ rel_2 \notin scope(r_1)\right)$

2. **Distributivität**

   a) $\exists r \in rel : (p(r) \bigvee q(r)) \iff \exists r_1 \in rel : p(r_1) \ \bigvee\ \exists r_2 \in rel : q(r_2)$

   b) $\forall r \in rel : (p(r) \bigwedge q(r)) \iff \forall r_1 \in rel : p(r_1) \ \bigwedge\ \forall r_2 \in rel : q(r_2)$

   c) $\exists r \in rel : (p(r) \bigwedge q(r)) \iff \exists r_1 \in rel : p(r_1) \ \bigwedge\ \exists r_2 \in rel : q(r_2)$
   $\left(r \notin freeVar(p) \ \bigwedge\ r \notin freeVar(q)\right)$

d) $\forall r \in rel : (p(r) \bigvee q(r)) \iff \forall r_1 \in rel : p(r_1) \bigvee \forall r_2 \in rel : q(r_2)$

$$\Big(r \notin freeVar(p) \bigwedge r \notin freeVar(q)\Big)$$

3. **Negation**

a) $\neg \exists r \in rel : p(r) \iff \forall r \in rel : \neg p(r)$

b) $\neg \forall r \in rel : p(r) \iff \exists r \in rel : \neg p(r)$

4. **Quantorabsorption**

a) $\exists r \in rel : p(r) \iff$ if $rel = \emptyset$ then $false$ else $p(x)$

$$\Big(r \notin freeVar(p)\Big)$$

b) $\forall r \in rel : p(r) \iff$ if $rel = \emptyset$ then $true$ else $p(x)$

$$\Big(r \notin freeVar(p)\Big)$$

5. **Quantorherausziehen**

a) $p(x) \bigwedge \exists r \in rel : q(r) \iff \exists r \in rel : (p(x) \bigwedge q(r))$

b) $p(x) \bigvee \forall r \in rel : q(r) \iff \forall r \in rel : (p(x) \bigvee q(r))$

c) $p(x) \bigvee \exists r \in rel : q(r) \iff$ if $rel = \emptyset$ then $p(x)$ else $\exists r \in rel : (p(x) \bigvee q(r))$

d) $p(x) \bigwedge \forall r \in rel : q(r) \iff$ if $rel = \emptyset$ then $p(x)$ else $\forall r \in rel : (p(x) \bigwedge q(r))$

## C.2 Konstruktive Operationen

1. **Assoziativität**

a) $rel_1 \times (rel_2 \times rel_3) \iff (rel_1 \times rel_2) \times rel_3$

b) $rel_1 \bigcup (rel_2 \bigcup rel_3) \iff (rel_1 \bigcup rel_2) \bigcup rel_3$

c) $rel_1 \bowtie (rel_2 \bowtie rel_3) \iff (rel_1 \bowtie rel_2) \bowtie rel_3$

2. **Kommutativität**

a) $rel_1 \times rel_2 \iff rel_2 \times rel_1$

b) $rel_1 \bigcup rel_2 \iff rel_2 \bigcup rel_1$

c) $rel_1 \bowtie rel_2 \iff rel_2 \bowtie rel_1$

## C.3 Selektive Operationen

1. **Zusammenfassung von Projektionen**
   $\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(rel)) \iff \pi_{A_1,\ldots,A_n}(rel)$

   $$\Big(A_1,\ldots,A_n \subseteq B_1,\ldots,B_m\Big)$$

2. **Zusammenfassung von Selektionen**
   $\sigma_p(\sigma_q(rel)) \iff \sigma_{p \wedge q}(rel)$

3. **Vertauschen von Selektion und Projektion**
   $\pi_{A_1,\ldots,A_n}(\sigma_p(rel)) \iff \pi_{A_1,\ldots,A_n}(\sigma_p(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(rel)))$

   $$\Big(\forall B_i \in \{B_1,\ldots,B_m\} : B_i \notin \{A_1,\ldots,A_n\} \bigwedge \{B_1,\ldots,B_m\} = freeVar(p)\Big)$$

4. **Vertauschen von Selektion und Union**
   $\sigma_p(rel_1 \bigcup rel_2) \iff \sigma_p(rel_1) \bigcup \sigma_p(rel_2)$

5. **Vertauschen von Selektion und Mengendifferenz**
   $\sigma_p(rel_1 \setminus rel_2) \iff \sigma_p(rel_1) \setminus \sigma_p(rel_2)$

6. **Vertauschen von Selektion und Kartesischem Produkt**

   a) $\sigma_p(rel_1 \times rel_2) \iff \sigma_{p_{rel_1}}(rel_1) \times \sigma_{p_{rel_2}}(rel_2)$

   $$\Big(\forall x \in freeVar(p_{rel_1}) : x \in attr(rel_1) \bigwedge \forall y \in freeVar(p_{rel_2}) : y \in attr(rel_2)\Big)$$

   b) $\sigma_p(rel_1 \times rel_2) \iff \sigma_p(rel_1) \times rel_2$

   $$\Big(\forall x \in freeVar(p) : x \in attr(rel_1)\Big)$$

   c) $\sigma_p(rel_1 \times rel_2) \iff \sigma_p\left(\sigma_{p_{rel_1}}(rel_1) \times rel_2\right)$

   $$\Big(\forall x \in freeVar(p_{rel_1}) : x \in attr(rel_1)\Big)$$

7. **Vertauschen von Selektion und Join**
   $\sigma_p(rel_1 \bowtie rel_2) \iff \sigma_{p_{rel_1}}(rel_1) \bowtie \sigma_{p_{rel_2}}(rel_2)$

   $$\Big(\forall x \in freeVar(p_{rel_1}) : x \in attr(rel_1) \bigwedge \forall y \in freeVar(p_{rel_2}) : y \in attr(rel_2)\Big)$$

8. **Vertauschen von Projektion und Kartesischem Produkt**
   $\pi_{A_1,\ldots,A_n}(rel_1 \times rel_2) \iff \pi_{B_1,\ldots,B_m}(rel_1) \times \pi_{C_1,\ldots,C_k}(rel_2)$

   $$\Big(\{B_1,\ldots,B_m\} \bigcup \{C_1,\ldots,C_k\} = \{A_1,\ldots,A_n\} \bigwedge \{B_1,\ldots,B_m\} \subseteq attr(rel_1) \bigwedge \{C_1,\ldots,C_k\} \subseteq attr(rel_2)\Big)$$

9. **Vertauschen von Projektion und Union**
   $\pi_{A_1,\ldots,A_n}(rel_1 \bigcup rel_2) \iff \pi_{A_1,\ldots,A_n}(rel_1) \bigcup \pi_{A_1,\ldots,A_n}(rel_2)$
10. **Vertauschen von Projektion und Join**
    $\pi_{A_1,\ldots,A_n}(rel_1 \bowtie rel_2) \iff \pi_{A_1,\ldots,A_n}(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(rel_1) \bowtie \pi_{A_1,\ldots,A_n,C_1,\ldots,C_k}(rel_2))$

    $\Big(\{B_1,\ldots,B_m\} \subseteq attr(rel_1 \bigcap rel_1 \bowtie rel_2) \bigwedge \{C_1,\ldots,C_k\} \subseteq attr(rel_2 \bigcap rel_1 \bowtie rel_2)\Big)$

## C.4 Leere Relationen

1. $\sigma_p(\emptyset) \iff \emptyset$
2. $\pi_{A_1,\ldots,A_n}(\emptyset) \iff \emptyset$
3. $\exists r \in \emptyset : p(r) \iff false$
4. $\forall r \in \emptyset : p(r) \iff true$

## C.5 Konstantes Prädikat

1. $\exists r \in rel : false \iff false$
2. $\forall r \in rel : true \iff true$
3. $\exists r \in rel : true \iff$ if $rel = \emptyset$ then $false$ else $true$
4. $\forall r \in rel : false \iff$ if $rel = \emptyset$ then $true$ else $false$
5. $\sigma_{true}(rel) \iff rel$
6. $\sigma_{false}(rel) \iff \emptyset$

# D Dynamische Optimierung am Beispiel von „Türme von Hanoi“

Im folgenden werden der TL-Code, der daraus erzeugte TML-Code und der dynamisch optimierte TML-Code für das Beispiel aus Abschnitt 7.1 „*Türme von Hanoi*“ präsentiert. Aus Platzgründen werden nur die Funktionen aus den Moduln *hanoi* und *intStack* behandelt, während die aus der Tycoon-Standardbibliothek importierten Moduln *list* und *print* sowie die in der initialen Programmierumgebung definierten Bindungen von '+', ':=', '==', '! =', etc. ausgelassen werden.

Die dynamische Optimierung wird folgendermaßen auf die Funktion *towers* aus dem Modul *hanoi* aufgerufen:

```
optimize(hanoi.towers);
```

Das Beispiel beruht auf Stapelmanipulationen (Modul *intStack*), die ihrerseits durch Listen implementiert sind (Modul *list* aus der Tycoon-Standardbibliothek). Nach der dynamischen Optimierung von *towers* werden die meisten Funktionen aus *intStack* (bis auf *init*) expandiert, so daß sich die im Code verbliebenen *oid*s direkt auf die Listenfunktionen beziehen. Dementsprechend wird für die vollständig integrierten Funktionen nach der dynamischen Optimierung von *towers* keinen Code angegeben.

Die Messungen an diesem Beispiel haben eine Performanzsteigerung von 100 % der dynamisch optimierten bezüglich der nicht optimierten Version erwiesen (vgl. Abb. 7.1 auf Seite 111).

## D.1 TL-Code

**Modul *intStack*:**

```
module intStack

import list print

export

  Let T=list.T(Int)

  let error(s :String) = print.string(s)

  let new() :T = list.cons(0 list.new(:Int))

  let bottom(s :T) = list.head(s) == 0

  let get(s :T) = list.head(s)

  let push(i :Int var s :T) :Ok = s := list.cons(i s)

  let init(n :Int) :T = begin
    let var s = new()
    for i = n downto 1 do
      s := list.cons(i s)
    end
    s
  end

  let pop(var s :T) :Int =
    if list.head(s) != 0 then
      let temp = list.head(s)
      s := list.tail(s)
      temp
    else
      error("nothing to pop")
      0
    end

end;
```

**Modul *hanoi*:**

```
module hanoi

import intStack print arrayOp
```

```
export

  let stackRange = 3

  let t = arrayOp.new(:intStack.T stackRange+1  intStack.new())

  let var movesDone = 0

  let move(var s1, s2 :intStack.T) :Ok =
  if intStack.bottom(s2) orif {intStack.get(s1) < intStack.get(s2)} then
    intStack.push(intStack.pop(s1) s2)
    movesDone := movesDone + 1
  else
    print.string(" Error in Towers.")
  end

  let rec tower(i,j,count :Int) :Ok = begin
    if count == 1 then
      move(t[i] t[j])
    else
      let other = 6 - i - j
      tower(i other count-1)
      move(t[i] t[j])
      tower(other j count-1)
    end
  end

  let towers() :Ok = begin
    t[1] := intStack.init(14)
    t[2] := intStack.new()
    t[3] := intStack.new()
    movesDone := 0
    tower(1 2 14)
    let var maxMoves = 16383
    if movesDone != maxMoves then
      print.string("Error in Towers.\n")
    end
  end

end;
```

## D.2 TML-Code (nicht optimiert)

**intStack.new:** *<oid 0x012723b0>*

```
proc(c_427)
  (lambda(list_426)
```

```
    | : ([] list_426 6 cont(t_428)
    | : ([] list_426 2 cont(t_429)
    | : (t_429 cont(t_430)
    | : (t_428 0 t_430 cont(t_431)
    | : (c_427 t_431)))))
    | <oid 0x0126724c>)
```

**intStack.bottom:** *<oid 0x012723c4>*

```
proc(s_364 c_365)
  (lambda(list_362 ==_363)
  | : ([] list_362 7 cont(t_366)
  | : (t_366 s_364 cont(t_367)
  | : (==_363 t_367 0 cont(t_368)
  | : (c_365 t_368))))
  | <oid 0x0126724c>
  | <oid 0x0068771c>)
```

**intStack.get:** *<oid 0x012723dc>*

```
proc(s_354 c_355)
  (lambda(list_353)
  | : ([] list_353 7 cont(t_356)
  | : (t_356 s_354 cont(t_357)
  | : (c_355 t_357)))
  | <oid 0x0126724c>)
```

**intStack.push:** *<oid 0x012723f0>*

```
proc(i_222 s_223 soffset_224 c_225)
  (lambda(list_220 :=_221)
  | : ([] list_220 6 cont(t_226)
  | : ([] s_223 soffset_224 cont(t_227)
  | : (t_226 i_222 t_227 cont(t_228)
  | : (:=_221 s_223 soffset_224 t_228 cont(t_229)
  | : (c_225 t_229)))))
  | <oid 0x0126724c>
  | <oid 0x0068772c>)
```

**intStack.init:** *<oid 0x01272408>*

```
proc(n_453 c_454)
  (lambda(new_450 list_451 :=_452)
  | : (new_450 cont(t_455)
  | : (array t_455 cont(s_456)
  | : (lambda(forEnd_457)
  | : | . (Y proc(c_458 for_459)
  | : | . (c_458
```

```
      | : | . | cont()
      | : | . | : (for_459 n_453)
      | : | . | cont(i_460)
      | : | . | : (< i_460 1
      | : | . | : | cont()
      | : | . | : | . (forEnd_457)
      | : | . | : | cont()
      | : | . | : | . ([] list_451 6 cont(t_461)
      | : | . | : | . ([] s_456 0 cont(t_462)
      | : | . | : | . (t_461 i_460 t_462 cont(t_463)
      | : | . | : | . (:=_452 s_456 0 t_463 cont(t_464)
      | : | . | : | . (- i_460 1 cont(t_465)
      | : | . | : | . (for_459 t_465)))))))))
      | : | cont()
      | : | . ([] s_456 0 cont(t_466)
      | : | . (c_454 t_466)))))
      | <oid 0x012723b0>
      | <oid 0x0126724c>
      | <oid 0x0068772c>)
```

**intStack.pop:** *<oid 0x01272424>*

```
    proc(s_251 soffset_252 c_253)
      (lambda(list_247 !=_248 :=_249 error_250)
      | : (lambda(join_254)
      | : | . ([] list_247 7 cont(t_255)
      | : | . ([] s_251 soffset_252 cont(t_256)
      | : | . (t_255 t_256 cont(t_257)
      | : | . (!=_248 t_257 0 cont(t_258)
      | : | . (== t_258 true false
      | : | . | cont()
      | : | . | : ([] list_247 7 cont(t_259)
      | : | . | : ([] s_251 soffset_252 cont(t_260)
      | : | . | : (t_259 t_260 cont(temp_261)
      | : | . | : ([] list_247 8 cont(t_262)
      | : | . | : ([] s_251 soffset_252 cont(t_263)
      | : | . | : (t_262 t_263 cont(t_264)
      | : | . | : (:=_249 s_251 soffset_252 t_264 cont(t_265)
      | : | . | : (join_254 temp_261)))))))))
      | : | . | cont()
      | : | . | : (error_250 "nothing to pop" cont(t_266)
      | : | . | : (join_254 0)))))))
      | : | cont(t_267)
      | : | . (c_253 t_267))
      | <oid 0x0126724c>
      | <oid 0x0068770c>
      | <oid 0x0068772c>
      | <oid 0x0127239c>)
```

**hanoi.move:** *<oid 0x012754c4>*

```
proc(s1_177 s1offset_178 s2_179 s2offset_180 c_181)
  (lambda(intStack_171 <_172 movesDone_173 +_174 :=_175 print_176)
  | : (lambda(join_182)
  | : | . (lambda(c_183)
  | : | . | : ([] intStack_171 3 cont(t_184)
  | : | . | : ([] s2_179 s2offset_180 cont(t_185)
  | : | . | : (t_184 t_185 cont(t_186)
  | : | . | : (== t_186 true false
  | : | . | : | cont()
  | : | . | : | . (c_183 true)
  | : | . | : | cont()
  | : | . | : | . ([] intStack_171 4 cont(t_187)
  | : | . | : | . ([] s1_177 s1offset_178 cont(t_188)
  | : | . | : | . (t_187 t_188 cont(t_189)
  | : | . | : | . ([] intStack_171 4 cont(t_190)
  | : | . | : | . ([] s2_179 s2offset_180 cont(t_191)
  | : | . | : | . (t_190 t_191 cont(t_192)
  | : | . | : | . (<_172 t_189 t_192 cont(t_193)
  | : | . | : | . (c_183 t_193))))))))))))
  | : | . | cont(t_194)
  | : | . | : (== t_194 true false
  | : | . | : | cont()
  | : | . | : | . ([] intStack_171 5 cont(t_195)
  | : | . | : | . ([] intStack_171 7 cont(t_196)
  | : | . | : | . (t_196 s1_177 s1offset_178 cont(t_197)
  | : | . | : | . (t_195 t_197 s2_179 s2offset_180 cont(t_198)
  | : | . | : | . ([] movesDone_173 0 cont(t_199)
  | : | . | : | . (+_174 t_199 1 cont(t_200)
  | : | . | : | . (:=_175 movesDone_173 0 t_200 cont(t_201)
  | : | . | : | . (join_182 t_201))))))))
  | : | . | : | cont()
  | : | . | : | . ([] print_176 6 cont(t_202)
  | : | . | : | . (t_202 " Error in Towers." cont(t_203)
  | : | . | : | . (join_182 t_203)))))
  | : | cont(t_204)
  | : | . (c_181 t_204))
  | <oid 0x01272444>
  | <oid 0x0068781c>
  | <oid 0x012754b8>
  | <oid 0x006877dc>
  | <oid 0x0068772c>
  | <oid 0x00d17b58>)
```

**hanoi.tower:** *<oid 0x012754ec>*

```
proc(i_120 j_121 count_122 c_123)
```

```
(lambda(tower_114 ==_115 t_116 move_117
        _raiseIndexOutOfBoundsError_118 -_119)
| : (lambda(join_124)
| : | . (==_115 count_122 1 cont(t_125)
| : | . (== t_125 true false
| : | . | cont()
| : | . | : (lambda(fail_126)
| : | . | : | . (< i_120 0
| : | . | : | . | cont()
| : | . | : | . | : (fail_126)
| : | . | : | . | cont()
| : | . | : | . | : (size t_116 cont(size_127)
| : | . | : | . | : (>= i_120 size_127
| : | . | : | . | : | cont()
| : | . | : | . | : | . (fail_126)
| : | . | : | . | : | cont()
| : | . | : | . | : | . (lambda(fail_128)
| : | . | : | . | : | . | : (< j_121 0
| : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . (fail_128)
| : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . (size t_116 cont(size_129)
| : | . | : | . | : | . | : | . (>= j_121 size_129
| : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : (fail_128)
| : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : (move_117 t_116 i_120
                                          t_116 j_121 cont(t_130)
| : | . | : | . | : | . | : | . | : (join_124 t_130)))))
| : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : (_raiseIndexOutOfBoundsError_118 23 18
                               "benchmarks/hanoi.tm" t_116 j_121
| : | . | : | . | : | . | : | cont(ignore_131)
| : | . | : | . | : | . | : | . (raise nil))))))
| : | . | : | cont()
| : | . | : | . (_raiseIndexOutOfBoundsError_118 23 13
                    "benchmarks/hanoi.tm" t_116 i_120
| : | . | : | . | cont(ignore_132)
| : | . | : | . | : (raise nil)))
| : | . | cont()
| : | . | : (-_119 6 i_120 cont(t_133)
| : | . | : (-_119 t_133 j_121 cont(other_134)
| : | . | : (-_119 count_122 1 cont(t_135)
| : | . | : (tower_114 i_120 other_134 t_135 cont(t_136)
| : | . | : (lambda(fail_137)
| : | . | : | . (< i_120 0
| : | . | : | . | cont()
| : | . | : | . | : (fail_137)
```

```
| : | . | : | . | cont()
| : | . | : | . | : (size t_116 cont(size_138)
| : | . | : | . | : (>= i_120 size_138
| : | . | : | . | : | cont()
| : | . | : | . | : | . (fail_137)
| : | . | : | . | : | cont()
| : | . | : | . | : | . (lambda(fail_139)
| : | . | : | . | : | . | : (< j_121 0
| : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . (fail_139)
| : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . (size t_116 cont(size_140)
| : | . | : | . | : | . | : | . | : | . (>= j_121 size_140
| : | . | : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : | . | : (fail_139)
| : | . | : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : | . | : (move_117 t_116 i_120
                                                    t_116 j_121 cont(t_141)
| : | . | : | . | : | . | : | . | : | . | : (-_119 count_122 1 cont(t_142)
| : | . | : | . | : | . | : | . | : | . | : (tower_114 other_134 j_121
                                                     t_142 cont(t_143)
| : | . | : | . | : | . | : | . | : | . | : (join_124 t_143)))))))
| : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : (_raiseIndexOutOfBoundsError_118 27 18
                                     "benchmarks/hanoi.tm" t_116 j_121
| : | . | : | . | : | . | : | . | : | cont(ignore_144)
| : | . | : | . | : | . | : | . | : | . (raise nil))))))
| : | . | : | cont()
| : | . | : | . (_raiseIndexOutOfBoundsError_118 27 13
                       "benchmarks/hanoi.tm" t_116 i_120
| : | . | : | . | cont(ignore_145)
| : | . | : | . | : (raise nil)))))))))
| : | cont(t_146)
| : | . (c_123 t_146))
| <oid 0x012754ec>
| <oid 0x0068771c>
| <oid 0x012754a0>
| <oid 0x012754c4>
| <oid 0x00687680>
| <oid 0x006877cc>)
```

**hanoi.towers:** `proc(c_14)`

```
(lambda(t_6 intStack_7 :=_8 movesDone_9 tower_10 !=_11
        print_12 _raiseIndexOutOfBoundsError_13)
| : (lambda(fail_15)
| : | . (< 1 0
| : | . | cont()
| : | . | : (fail_15)
```

```
| : | . | cont()
| : | . | : (size t_6 cont(size_16)
| : | . | : (>= 1 size_16
| : | . | : | cont()
| : | . | : | . (fail_15)
| : | . | : | cont()
| : | . | : | . ([] intStack_7 6 cont(t_17)
| : | . | : | . (t_17 14 cont(t_18)
| : | . | : | . (:=_8 t_6 1 t_18 cont(t_19)
| : | . | : | . (lambda(fail_20)
| : | . | : | . | : (< 2 0
| : | . | : | . | : | cont()
| : | . | : | . | : | . (fail_20)
| : | . | : | . | : | cont()
| : | . | : | . | : | . (size t_6 cont(size_21)
| : | . | : | . | : | . (>= 2 size_21
| : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : (fail_20)
| : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : ([] intStack_7 2 cont(t_22)
| : | . | : | . | : | . | : (t_22 cont(t_23)
| : | . | : | . | : | . | : (:=_8 t_6 2 t_23 cont(t_24)
| : | . | : | . | : | . | : (lambda(fail_25)
| : | . | : | . | : | . | : | . (< 3 0
| : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : (fail_25)
| : | . | : | . | : | . | : | . | cont()
| : | . | : | . | : | . | : | . | : (size t_6 cont(size_26)
| : | . | : | . | : | . | : | . | : (>= 3 size_26
| : | . | : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . | : | . (fail_25)
| : | . | : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . | : | . ([] intStack_7 2 cont(t_27)
| : | . | : | . | : | . | : | . | : | . (t_27 cont(t_28)
| : | . | : | . | : | . | : | . | : | . (:=_8 t_6 3 t_28 cont(t_29)
| : | . | : | . | : | . | : | . | : | . (:=_8 movesDone_9 0 0 cont(t_30)
| : | . | : | . | : | . | : | . | : | . (tower_10 1 2 14 cont(t_31)
| : | . | : | . | : | . | : | . | : | . (array 16383 cont(maxMoves_32)
| : | . | : | . | : | . | : | . | : | . (lambda(join_33)
| : | . | : | . | : | . | : | . | : | . | : ([] movesDone_9 0 cont(t_34)
| : | . | : | . | : | . | : | . | : | . | : ([] maxMoves_32 0 cont(t_35)
| : | . | : | . | : | . | : | . | : | . | : (!=_11 t_34 t_35 cont(t_36)
| : | . | : | . | : | . | : | . | : | . | : (== t_36 true false
| : | . | : | . | : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . | : | . | : | . ([] print_12 6 cont(t_37)
| : | . | : | . | : | . | : | . | : | . | : | . (t_37 "Error in Towers.\n"
                                                        cont(t_38)
| : | . | : | . | : | . | : | . | : | . | : | . (join_33 t_38)))
```

```
| : | . | : | . | : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . | : | . | : | . (join_33 nil)))))
| : | . | : | . | : | . | : | . | : | . | : | . | cont(t_39)
| : | . | : | . | : | . | : | . | : | . | : | . | : (c_14 t_39))))))))))))
| : | . | : | . | : | . | : | . | : | cont()
| : | . | : | . | : | . | : | . | : | . (_raiseIndexOutOfBoundsError_13 35 6
                                        "benchmarks/hanoi.tm" t_6 3
| : | . | : | . | : | . | : | . | : | . | cont(ignore_40)
| : | . | : | . | : | . | : | . | : | . | : (raise nil)))))))))
| : | . | : | . | : | . | cont()
| : | . | : | . | : | : (_raiseIndexOutOfBoundsError_13 34 6
                            "benchmarks/hanoi.tm" t_6 2
| : | . | : | . | : | : | cont(ignore_41)
| : | . | : | . | : | : | . (raise nil)))))))))
| : | cont()
| : | . (_raiseIndexOutOfBoundsError_13 33 6 "benchmarks/hanoi.tm" t_6 1
| : | . | cont(ignore_42)
| : | . | : (raise nil)))
| <oid 0x012754a0>
| <oid 0x01272444>
| <oid 0x0068772c>
| <oid 0x012754b8>
| <oid 0x012754ec>
| <oid 0x0068770c>
| <oid 0x00d17b58>
| <oid 0x00687680>)
```

## D.3 TML-Code nach der dynamischen Optimierung

**intStack.new:** *<oid 0x012723b0>*

▷ *vollständig im Code integriert*

**intStack.bottom:** *<oid 0x012723c4>*

▷ *vollständig im Code integriert*

**intStack.get:** *<oid 0x012723dc>*

▷ *vollständig im Code integriert*

**intStack.push:** *<oid 0x012723f0>*

▷ *vollständig im Code integriert*

**intStack.init:** *<oid 0x01272408>*

```
proc(n_453 c_454)
  (vector 1 cont(t_475)
  (vector 2 0 t_475 cont(t_455)
  (array t_455 cont(s_456)
  (Y proc(c_458 for_459)
  (c_458
  | cont()
  | : (for_459 n_453)
  | cont(i_460)
  | : (< i_460 1
  | : | cont()
  | : | . ([] s_456 0 c_454)
  | : | cont()
  | : | . ([] s_456 0 cont(t_462)
  | : | . (vector 2 i_460 t_462 cont(t_463)
  | : | . ([]:= s_456 0 t_463 cont()
  | : | . (- i_460 1 for_459))))))))))
```

**intStack.pop:** *<oid 0x01272424>*

▷ *vollständig im Code integriert*

**hanoi.move:** *<oid 0x012754c4>*

```
proc(s1_177 s1offset_178 s2_179 s2offset_180 c_181)
  (lambda(c_183)
  | : ([] s2_179 s2offset_180 cont(t_185)
  | : (<oid 0x012671fc> t_185 cont(t_377)
  | : (== t_377 0
  | : | cont()
  | : | . (c_183 true)
  | : | cont()
  | : | . ([] s1_177 s1offset_178 cont(t_188)
  | : | . (<oid 0x012671fc> t_188 cont(t_189)
  | : | . ([] s2_179 s2offset_180 cont(t_191)
  | : | . (<oid 0x012671fc> t_191 cont(t_192)
  | : | . (< t_189 t_192
  | : | . | cont()
  | : | . | : (c_183 true)
  | : | . | cont()
  | : | . | : (c_183 false)))))))))
  | cont(t_194)
  | : (== t_194 true false
  | : | cont()
  | : | . (lambda(c_337)
  | : | . | : ([] s1_177 s1offset_178 cont(t_338)
```

```
| : | . | : (<oid 0x012671fc> t_338 cont(t_339)
| : | . | : (== t_339 0
| : | . | : | cont()
| : | . | : | . (<oid 0x00d10ef0> 1 "nothing to pop" cont(t_340)
| : | . | : | . (c_337 0))
| : | . | : | cont()
| : | . | : | . ([] s1_177 s1offset_178 cont(t_341)
| : | . | : | . (<oid 0x012671fc> t_341 cont(temp_342)
| : | . | : | . ([] s1_177 s1offset_178 cont(t_343)
| : | . | : | . (<oid 0x012671e8> t_343 cont(t_344)
| : | . | : | . ([]:= s1_177 s1offset_178 t_344 cont()
| : | . | : | . (c_337 temp_342)))))))))
| : | . | cont(t_197)
| : | . | : ([] s2_179 s2offset_180 cont(t_245)
| : | . | : (vector 2 t_197 t_245 cont(t_246)
| : | . | : ([]:= s2_179 s2offset_180 t_246 cont()
| : | . | : ([] <oid 0x012754b8> 0 cont(t_199)
| : | . | : (+ t_199 1 cont(t_200)
| : | . | : ([]:= <oid 0x012754b8> 0 t_200 cont()
| : | . | : (c_181 nil)))))))))
| : | cont()
| : | . (<oid 0x00d10ef0> 1 " Error in Towers." c_181)))
```

**hanoi.tower:** *<oid 0x012754ec>*

```
proc(i_120 j_121 count_122 c_123)
  (lambda(c_417)
  | : (== count_122 1
  | : | cont()
  | : | . (c_417 true)
  | : | cont()
  | : | . (c_417 false))
  | cont(t_125)
  | : (== t_125 true false
  | : | cont()
  | : | . (lambda(fail_126)
  | : | . | : (< i_120 0 fail_126 cont()
  | : | . | : (>= i_120 4 fail_126 cont()
  | : | . | : (lambda(fail_128)
  | : | . | : | . (< j_121 0 fail_128 cont()
  | : | . | : | . (>= j_121 4 fail_128 cont()
  | : | . | : | . (<oid 0x012754c4> <oid 0x012754a0> i_120 <oid 0x012754a0>
                      j_121 c_123)))
  | : | . | : | cont()
  | : | . | : | . (vector "indexOutOfBoundsError" 23 18
                      "benchmarks/hanoi.tm" <oid 0x012754a0>
  | : | . | : | . | j_121
  | : | . | : | . | cont(t_170)
```

```
    | : | . | : | . | : (raise t_170)))))
    | : | . | cont()
    | : | . | : (vector "indexOutOfBoundsError" 23 13
                    "benchmarks/hanoi.tm" <oid 0x012754a0>
    | : | . | : | i_120
    | : | . | : | cont(t_163)
    | : | . | : | . (raise t_163)))
    | : | cont()
    | : | . (- 6 i_120 cont(t_133)
    | : | . (- t_133 j_121 cont(other_134)
    | : | . (- count_122 1 cont(t_135)
    | : | . (<oid 0x012754ec> i_120 other_134 t_135 cont(t_136)
    | : | . (lambda(fail_137)
    | : | . | : (< i_120 0 fail_137 cont()
    | : | . | : (>= i_120 4 fail_137 cont()
    | : | . | : (lambda(fail_139)
    | : | . | : | . (< j_121 0 fail_139 cont()
    | : | . | : | . (>= j_121 4 fail_139 cont()
    | : | . | : | . (<oid 0x012754c4> <oid 0x012754a0> i_120 <oid 0x012754a0>
                        j_121 cont(t_141)
    | : | . | : | . (- count_122 1 cont(t_142)
    | : | . | : | . (<oid 0x012754ec> other_134 j_121 t_142 c_123)))))
    | : | . | : | cont()
    | : | . | : | . (vector "indexOutOfBoundsError" 27 18
                        "benchmarks/hanoi.tm" <oid 0x012754a0>
    | : | . | : | . | j_121
    | : | . | : | . | cont(t_402)
    | : | . | : | . | : (raise t_402)))))
    | : | . | cont()
    | : | . | : (vector "indexOutOfBoundsError" 27 13
                    "benchmarks/hanoi.tm" <oid 0x012754a0>
    | : | . | : | i_120
    | : | . | : | cont(t_395)
    | : | . | : | . (raise t_395)))))))))
```

**hanoi.towers:** *<oid 0x012723c4>*

```
proc(c_14)
  (<oid 0x01272408> 14 cont(t_18)
  ([]:= <oid 0x012754a0> 1 t_18 cont()
  (vector 1 cont(t_445)
  (vector 2 0 t_445 cont(t_23)
  ([]:= <oid 0x012754a0> 2 t_23 cont()
  (vector 1 cont(t_439)
  (vector 2 0 t_439 cont(t_28)
  ([]:= <oid 0x012754a0> 3 t_28 cont()
  ([]:= <oid 0x012754b8> 0 0 cont()
  (<oid 0x012754ec> 1 2 14 cont(t_31)
```

```
(array 16383 cont(maxMoves_32)
([] <oid 0x012754b8> 0 cont(t_34)
([] maxMoves_32 0 cont(t_35)
(== t_34 t_35
| cont()
| : (c_14 nil)
| cont()
| : (<oid 0x00d10ef0> 1 "Error in Towers.\n" c_14)))))))))))))))))
```

# Literaturverzeichnis


[Abiteboul, Beeri 92] S. Abiteboul, C. Beeri, *On the Power of Languages for the Manipulation of Complex Objects*, Technical Report, 1992.

[Aho et al. 86] A. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools* , Addison-Wesley Publishing Company, 1986.

[Allen, Cocke 72] F. E. Allen, J. Cocke, *A Catalogue of Optimizing Transformations*, in *Design and Optimization of Compilers*, edited by R. Rustin, Prentice Hall, 1972.

[Appel 89] A. Appel, *Continuation-Passing, Closure-Passing Style*, ACM SIGPLAN Notices 2/89, 1989.

[Appel 92] A. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

[Beeri, Kornatzki 91] C. Beeri, Y. Kornatzki, *Algebraic Optimization of Object-Oriented Query Languages*, Report, The Hebrew University, Jerusalem, 1991.

[Bertino, Guglielmina 93] E. Bertino, C. Guglielmina, *Path Index: An Approach to the Efficient Execution of Object-Oriented Queries*, Data & Knowledge Engeneering 1/93, 1993.

[Cardelli 90] L. Cardelli, *The Quest Language and System*, Tracking Draft, DEC SRC, Palo Alto, 1990.

[Chambers 92] C. Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Languages*, Ph.D. Thesis, Stanford University, 1992.

[Chambers, Ungar 90] C. Chambers, D. Ungar, *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs*, ACM SIGPLAN Notices 6/90, 1990.

[Chambers, Ungar 91] C. Chambers, D. Ungar, *Making Pure Object-Oriented Languages Practical*, Report, Stanford University, 1991.

[Chang et al. 92] P. P. Chang, S. A. Mahlke, W. Y. Chen, W.-M. W. Hwu, *Profile-Guided Automatic Inline Expansion for C Programs*, Software - Practice and Experience 5/92, 1992.

[Cooper et al. 91] K. D. Cooper, M. W. Hall, L. Torczon, *An Experiment with Inline Substitution*, Software - Practice and Experience 6/91, 1991.

[Davidson 86] J. Davidson, *A Retargetable Instruction Reorganizer*, ACM SIGPLAN Notices 7/86, 1986.

[Davidson, Holler 88] J. W. Davidson, A. M. Holler, *A Study of a C Function Inliner*, Software - Practice and Experience 8/88, 1988.

[Dean, Chambers 93] J. Dean, C. Chambers, *Training Compilers for Better Inlining Decisions*, Research Paper, University of Washington, 1993.

[Dearle et al. 89] A. Dearle, R. Connor, F. Brown, R. Morisson, *Napier88 - A Database Programming Language?*, Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon, Juni 1989.

[Dhamdhere et al. 92] D. M. Dhamdhere, B. K. Rosen, F. K. Zadeck, *How to Analyze Large Programs Efficiently and Informatively*, ACM SIGPLAN Notices 7/92, 1992.

[Dörfler, Peschek 88] W. Dörfler, W. Peschek, *Einführung in die Mathematik für Informatiker*, Carl Hanser Verlag, 1988.

[Drechsler, Stadel 93] K.-H. Drechsler, M. P. Stadel, *A Variation of Knoop, Rüthing, and Steffens's Lazy Code Motion*, ACM SIGPLAN Notices 5/93, 1993.

[Fegaras, Stemple 91] L. Fegaras, D. Stemple, *Using Type Transformation in Database System Implementation*, Report, University of Massachusetts, 1991.

[Fisher, LeBlanc 88] C. Fisher, R. LeBlanc, *Crafting a Compiler*, Benjamin/Cummings Publishing Company, 1988.

[Flanagan et al. 93] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, *The Essence of Compiling with Continuations*, ACM SIGPLAN Notices 6/93, 1993.

[Freytag 87] J. C. Freytag, *Translating Relational Queries into Iterative Programs*. Springer Verlag, 1987.

[Gawecki, Matthes 94] A. Gawecki, F. Matthes, *The Tycoon Machine Language TML: An Optimizable Persistent Program Representation*, Report, Hamburg University, 1994.

[Gawecki 91] A. Gawecki, *Ein optimierender Übersetzer für Smalltalk unter Verwendung der Techniken der empfängerspezifischen Übersetzung und des Programmierens mit Fortsetzungen*, Diplomarbeit, Universität Hamburg, 1991.

[Gifford, Lucassen 86] D. K. Gifford, J. M. Lucassen, *Integrating Functional and Imperative Programming*, ACM SIGPLAN Notices 4/86, 1986.

[Hölzle, Ungar 94] U. Hölzle, D. Ungar, *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*, ACM SIGPLAN Notices 6/94, 1994.

[Hwu, Chang 89] W. W. Hwu, P. P. Chang, *Inline Function Expansion for Compiling C Programs*, ACM SIGPLAN Notices 7/89, 1989.

[Hyun, Doberkat 85] K. C. Hyun, E.-E. Doberkat, *Inline Expansion of SETL Procedures*, Report, Hochschule Hildesheim, 1985.

[Ingres 90] Ingres Corp., *Language Reference Manual for INGRES/Windows 4GL for the UNIX and VMS Operating Systems*, Ingres Corp., Alameda, 1990.

[Juhacz 94] D. M. Juhacz, *Query and Bulk Type Extensions in Higher-Order Polymorphic Languages*, Master's Thesis, Hamburg University, 1994.

[Kelsey 89] R. A. Kelsey, *Compilation by Program Transformation*, Technical Report, Yale University, 1989.

[Kelsey, Hudak 89] R. A. Kelsey, P. Hudak, *Realistic Compilation by Program Transformation*, ACM SIGPLAN Notices 2/89, 1989.

[Kessler et al. 86] R.R. Kessler, J.C. Peterson, H. Carr, G.P. Duggan, J. Knell, J.J. Krohnfeldt, *EPIC - A Retargetable, Highly Optimizing Lisp Compiler*, ACM SIGPLAN Notices 7/86, 1986.

[Kim et al. 84] W. Kim, D.S. Reiner, D.S. Batory, *Query Processing in Database Systems*, Springer-Verlag, 1984.

[Klein 90] H. Klein, *Prädikatevaluation in einem typvollständigen Datenmodell*, Diplomarbeit, J. W. Goethe-Universität, Frankfurt/Main, 1990.

[Knoop et al. 92] J. Knoop, O. Rüthing, B. Steffen, *Lazy Code Motion*, ACM SIGPLAN Notices 7/92, 1992.

[Koch 85] J. Koch, *Relationale Anfragen: Zerlegung und Optimierung*, Springer Verlag, 1985.

[Kranz et al. 86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, N. Adams, *Orbit: An Optimizing Compiler for Scheme*, ACM SIGPLAN Notices, 21(7):219–233, July 1986.

[Landi et al. 93] W. Landi, B. G. Ryder. S. Zhang, *Interprocedural Modification Side Effect Analysis with Pointer Aliasing*, ACM SIGPLAN Notices 6/93, 1986.

[Lockeman, Schmidt 87] P. Lockeman, J. W. Schmidt, *Datenbankhandbuch*, Springer Verlag, 1987.

[Maslo, Dittrich 93] P. Maslo, S. Dittrich, *Das große Buch zu VisualBASIC 3.0 für Windows*, Data Becker, Düsseldorf, 1993.

[Mathiske 92] B. Mathiske, *Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung*, Diplomarbeit, Universität Hamburg, 1992.

[Matthes 92a] F. Matthes, *Generische Datenbankprogrammierung: Sprachliche und architektonische Grundlagen*, Dissertation, Universität Hamburg, 1992.

[Matthes 92b] F. Matthes, *Preliminary Definition of the Tycoon Language TL*, Report, Hamburg University, 1992.

[Matthes 93] F. Matthes, *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*, Springer Verlag, 1993.

[Matthes et al. 94] F. Matthes, S. Müßig, J. W. Schmidt, *Persistent Polymorphic Programming in Tycoon: An Introduction*, Technical Report, Hamburg University, 1994.

[Matthes, Schmidt 91] F. Matthes, J. W. Schmidt, *Bulk Types: Built-In or Add-On?*, Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece, August 27-30, 1991.

[Mielinski 91] R. Mielinski, *Anfrageoptimierung in einem Non-Standard Datenbanksystem*, Diplomarbeit, Universität Hamburg, 1991.

[MIPS] MIPS Inc., *MIPS C Compiler Reference Manual.*

[Müller 91] R. Müller, *Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung*, Diplomarbeit, J. W. Goethe-Universität, Frankfurt/Main, 1991.

[Niederee 92] C. Niederee, *Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung*, Diplomarbeit, Universität Hamburg, 1992.

[Orr et al. 93] D. B. Orr, R. W. Mecklenburg, P. J. Hoogenboom, J. Leprean, *Dynamic Program Monitoring and Transformation Using the OMOS Object Server*, Proceedings of the 26th Hawaii International Conference on System Sciences, January 5-8, 1993.

[Pendergrast, Ryder 86] J. S. Pendergrast, B. G. Ryder, *A Globally Optimizing Compiler for FP*, Technical Report, Rutgers University, 1986.

[Peyton-Jones 87] S. L. Peyton-Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[Plotkin 81] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report, Aarhus University, 1981.

[Schmidt 77] J. W. Schmidt, *Some High Level Language Constructs for Data of Type Relation*, ACM SIGPLAN Notices 8/77, 1977.

[Schmidt, Matthes 92] J. W. Schmidt, F. Matthes, *The Database Programming Language DBPL - Rational and Report*, Technical Report, Hamburg University, 1992.

[Schmidt, Matthes 93] J. W. Schmidt, F. Matthes, *Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems*, Technical Report, Hamburg University, 1993.

[Schröder 93] G. Schröder, *Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianzen*, Diplomarbeit, Universität Hamburg, 1993.

[Solaris] Solaris 2.2, *SunOs$^{TM}$ 5.2 Reference Manual. Library Routines*, Sun Microsystems Inc., 1993.

[Srivastava, Wall 92] A. Srivastava, D. Wall, *A Practical System for Intermodule Code Optimization at Link Time*, DEC WRL Research Report 92/6, 1992.

[Steele 78] G. L. Steele, *Rabbit: A Compiler for Scheme*, Ph.D. Thesis, Massachusetts Institute of Technology, 1978.

[Subieta 91] K. Subieta, *Architecture of Database Systems: The Ingres Case*, Lecture Notes, Hamburg University, 1991.

[Teodosiu 91] D. Teodosiu, *HARE: An Optimizing Portable Compiler for Scheme*, ACM SIGPLAN Notices 2/91, 1991.

[Tjiang, Hennessy 92] S. W. K. Tjiang, J. L. Hennessy, *Sharlit - A Tool for Building Optimizers*, ACM SIGPLAN Notices 7/92, 1992.

[Ullman 88a] J. D. Ullman, *Database and Knowledge-Base Systems, Vol. I*, Computer Science Press, 1988.

[Ullman 88b] J. D. Ullman, *Database and Knowledge-Base Systems, Vol. II: The New Technologies*, Computer Science Press, 1988.

[Wolfe 92] M. Wolfe, *Beyond Induction Variables*, ACM SIGPLAN Notices 7/92, 1992.

[Wulf et al. 73] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, *The Design of an Optimizing Compiler*, Report, Carnegie-Mellon-University, 1973.