Optimierung der persistenten objektorientierten Programmiersprache TooL

Diplomarbeit von
Martin Pakendorf
Arbeitsbereich Datenbanken und Informationssysteme
Fachbereich Informatik
Universität Hamburg

27. Juni 1996

eingereicht bei
Prof. Dr. Joachim W. Schmidt
und
Dr. Martin Lehmann

Inhaltsverzeichnis

1. Einleitung **1**

1.1 Gliederung 2

2. Die Sprache TooL in der persistenten Programmierumgebung Tycoon **3**

2.1 TML: eine maschinenunabhängige Zwischensprache 4

2.2 TVM: eine portable virtuelle Maschine 5

2.3 TSP: eine portable Objektspeicherschnittstelle 5

2.4 TooL im Tycoon System 6

3. Die Programmiersprache TooL **7**

3.1 Objektorientierte Programmierung 7

3.2 Klassen und Vererbung 7

3.3 Vererbung und der Aufruf von Methoden 9

3.4 Reine Objektorientierung 10

3.5 Das TooL Typsystem 12

3.5.1 Strukturelle Typisierung 12

3.5.2 Der Typ Self 12

3.5.3 Subtypisierung 13

3.5.4 Die Ähnlichkeitsrelation zwischen Objekttypen 14

3.5.5 Subtypbegrenzung und Ähnlichkeitsbegrenzung 15

3.5.6 Typparameter 15

4. Ansätze zur Optimierung objektorientierter Sprachen **16**

4.1 Klassenabschätzung 16

4.2 Inline Caching 17

4.3 Klassenanalyse und Splitting 18

4.4 Profilgesteuerte Optimierung 19

4.5 Empfängerspezifische Übersetzung 21

5. Lambdakalkül im Fortsetzungsstil als Zwischensprache **22**
5.1 TML . . . 23
5.2 Die Transformation von TooL nach TML . . . 26
5.3 Reduktion als grundlegendes Optimierungprinzip von CPS . . . 28
5.3.1 Beta-Reduktion . . . 28

6. Der TooL-Optimierer **31**
6.1 Profiling . . . 32
6.2 Aufbau eines empfänger- und argumentspezifisch optimierten Methodencaches 33
6.3 Markierte Typen . . . 33
6.4 Klassenanalyse . . . 35
6.4.1 Der Analyse-Algorithmus . . . 35
6.4.2 Ein Beispiel . . . 36
6.4.3 Verbreitung von Klasseninformation . . . 37
6.4.4 Optimierung durch Beta-Reduktion . . . 38
6.4.5 Aufruf einer spezialisierten Methode . . . 39
6.5 Elimination von Endrekursion . . . 40

7. Codegenerierung für eine virtuelle Registermaschine **46**
7.1 Die Architektur der Tycoon Registermaschine . . . 47
7.2 Codegenerierung für die Registermaschine . . . 49
7.2.1 Applikationen . . . 49
7.2.2 Abstraktionen . . . 50
7.3 Registerallokation . . . 53
7.3.1 Lebendigkeitsanalyse . . . 54
7.3.2 Registerzuordnung . . . 55
7.3.3 Zuteilung der Register . . . 56

8. Quantitative Ergebnisse **59**
8.1 Die Performanz von TooL . . . 59
8.1.1 Performanzsteigerungen durch Optimierungen . . . 60
8.1.2 Codezuwachs durch Optimierungen . . . 62
8.1.3 TooL im Vergleich zu TL und Java . . . 63
8.1.4 Fazit . . . 64
8.2 Performanzvergleich zwischen der Registerarchitektur und der Kellerarchitektur 65
8.2.1 Auswirkung der verschiedenen Interpreterimplementierungen . . . 65
8.2.2 Benchmarks . . . 65
8.2.3 Aufwand der Registerallokation . . . 66
8.2.4 Fazit . . . 67

9. Zusammenfassung und Ausblick **68**

A. Optimierungsphasen am Beispiel der while-Methode **70**

B. TML-Primitive **73**

B.1 Ursprüngliche TML-Primitive im Tycoon-Backend 73

B.2 TooL-spezifische TML-Primitive 74

C. TRM-Befehlssatz **75**

Literaturverzeichnis **76**

1. Einleitung

Im Kern jedes Softwareentwurfs steht die Modellierung von Problemlösungen und deren Abbildung auf konkrete Mechanismen eines Informationssystems. Modellierung und Abbildung können dabei je nach Mechanismus mehr oder weniger konzeptuell voneinander getrennt sein. Als einer der Vorzüge eines objektorientierten Ansatzes gilt, daß ein objektorientiertes Modell direkt auf Elemente objektorientierter Programmiersprachen abgebildet werden kann. Modellierungs- und Abbildungsmechanismus besitzen den gleichen syntaktischen und semantischen Stamm, so daß Modell und Implementation stark gekoppelt sind. Der Grad der Vereinheitlichung von Modell und Abbildung wird von der Ausdrucksmächtigkeit der Programmiersprache und ihrer Nähe zur reinen Objektorientierung beeinflußt.

Rein objektorientierte Sprachen kommen allerdings in der Praxis seltener zum Einsatz als hybride Sprachen mit imperativen und objektorientierten Elementen, denn ihre Ausführungsgeschwindigkeit unterliegt einem aus der Wissensrepräsentation bekannten „*tradeoff between the expressiveness of a representational language and its computational tractability*“[Levesque, Brachman 85]. Die Ausdrucksmächtigkeit rein objektorientierter Sprachen wird erkauft durch eine schwache Performanz, die den Ansprüchen realer Anwendungen oft nicht genügt.

Fortschritte in Optimierungstechniken haben den Leistungsrückstand rein objektorientierter Sprachen in der jüngeren Zeit entschieden verringert; so wird für die Sprache SELF berichtet[Hölzle 94], daß SELF-Programme die halbe Ausführungsgeschwindigkeit von C++, einer der schnellsten hybriden objektorientierten Sprachen, erreichen.

Bemerkenswert ist dabei die Tatsache, daß die Optimierungstechniken im SELF-Übersetzer den rein objektorientierten Charakter von SELF über Optimierungen hinaus erhalten. Für die neu entwickelte, rein objektorientierte Sprache TooL[1] wurde im Rahmen der vorliegenden Diplomarbeit ein Optimierungsverfahren entwickelt, das ebenfalls unter Wahrung der rein objektorientierten Natur zu zwei- bis dreifach kürzeren Ausführungszeiten führt.

Zwei Ansätze zur Leistungssteigerung werden untersucht: die maschinenunabhängige Optimierung auf einer CPS-Zwischenrepräsentation, und die Möglichkeit, durch bessere Codegenerierung kürzere Laufzeiten auf einer virtuellen Maschine zu erzielen. Dabei zeigt sich, daß ein recht allgemein gehaltener Optimierungsansatz gute Ergebnisse erzielen kann und die negativen Effekte der reinen Objektorientierung eingedämmt werden können. Ein erster Leistungsvergleich mit der nicht-objektorientierten, verwandten Sprache TL[2] ergibt, daß optimierte TooL-Programme im Schnitt ein Drittel der Leistung von optimierten TL-Programmen erbringen. Da nicht alle Optimierungsmöglichkeiten rigoros ausgenutzt werden, besteht die Erwartung, dieses recht gute Resultat noch verbessern zu können.

[1] Tycoon object-oriented Language
[2] Tycoon Language

Weniger Auswirkungen auf die Leistung hat die Architektur einer interpretierenden virtuellen Maschine und die dazugehörige Codegenerierungsstrategie. Eine Erkenntnis aus diesem Teil der Arbeit ist, daß für den Einsatz einer virtuellen Maschine die aus der Mode gekommene Kellerarchitektur eine geeignete Ausführungsplattform ist, da diese recht einfache Codegenerierungsverfahren zuläßt.

1.1 Gliederung

Die Sprache TooL ist Bestandteil der persistenten Programmierumgebung Tycoon[3], deren wesentliche Merkmale in Kapitel 2 aufgeführt werden.

In Kapitel 3 werden neben einer kurzen Einführung in objektorientierte Begriffe zwei wichtige Aspekte von TooL behandelt: die Bedeutung und das Ausmaß der reinen Objektorientierung in TooL und das Typsystem von TooL, das eine Einschränkung durch die Optimierungstechnik der vorliegenden Arbeit erfährt.

Eines der Hauptziele der Optimierung objektorientierter Sprachen besteht darin, Klasseninformation zu gewinnen, um mit ihrer Hilfe dynamische Methodenaufrufe zu eliminieren. Die Techniken verschiedener Übersetzer werden in Kapitel 4 dargestellt, damit eine Diskussionsgrundlage für den Optimierungsansatz dieser Arbeit, der in Kapitel 6 vorgestellt wird, besteht.

Die Zwischensprache und Optimierungsrepräsentation der Übersetzer für TL und TooL ist eine Variante des Lambdakalküls im Fortsetzungsstil, die auch Grundlage der optimierenden Transformationen darstellt. Auf diese Zusammenhänge wird in Kapitel 5 eingegangen.

Die Frage einer Leistungsteigerung durch den Einsatz einer virtuellen Registerarchitektur mit der dazugehörigen Codegenerierung ist das Thema von Kapitel 7. Die Berichterstattung der Leistung der Registermaschine und der Optimierungsergebnisse findet in Kapitel 8 statt, gefolgt von einer Zusammenfassung und einem Ausblick in Kapitel 9, in dem die Erfahrungen dieser Arbeit bewertet werden.

[3] TYped Communicating Objects in Open eNvironments

2. Die Sprache TooL in der persistenten Programmierumgebung Tycoon

Durch informationstechnologische Entwicklungen der letzten Jahre werden neue Anforderungen an Programmiersprachen und Werkzeuge, die zur Umsetzung visionärer Anwendungen benötigt werden, gestellt. Für die Aufgaben der Persistenz von komplexen Objekten, Ausführungszuständen und Ressourcen in heterogenen verteilten Umgebungen, der Mobilität von Code und Daten, und letztlich der Portabilität von Programmen bietet das Tycoon-System einen integrierten und orthogonalen Ansatz. Im Gegensatz zu einer architektonischen Trennung in Datenbank, Programmiersprache und plattformabhängigem Übersetzer stellt Tycoon eine Einheit dar: jeder Wert eines Datentyps der Programmiersprachen TooL und TL (Tycoon Language) [Matthes, Schmidt 92] ist potentiell persistent, die Übersetzer erzeugen portablen Code einer virtuellen Maschine, und ein integriertes Laufzeitsystem dient als Schnittstelle zu Betriebssystem, Speicherverwaltung und Objektspeicher.

Die Programmiersprachen TooL und TL sind typisierte Sprachen höherer Ordnung mit parametrischem Polymorphismus und Subtyppolymorphismus. TL enthält funktionale und imperative Elemente, während TooL einen rein objektorientierten Ansatz (siehe Kapitel 3) verfolgt. In Verbindung mit der orthogonalen Persistenz jedes Objekts dienen TL und TooL als geeignete Modellierungsmittel für radikal neue Paradigmen der Informationsverarbeitung wie mobile Agenten [Mathiske et al. 95], persistente Threads [Matthes, Schmidt 94] oder Workflow-Automaten. Diese Paradigmen beruhen auf der Tatsache, daß Funktionen samt Bindungen Datenobjekte erster Klasse und damit persistent speicherbar, als aktive Nachrichten über Netzwerke austauschbar und aufgrund des portablen Codeformates auf verschiedenen Rechnerarchitekturen ausführbar sind.

Die Schichtenarchitektur des Tycoon-Systems in Abbildung 2.1 ähnelt dem allgemeinen Schema eines Übersetzers, das eine Aufteilung in Parser und Typprüfer (*front end*, Analysephase) einerseits, und Optimierer und Codegenerator (*back end*, Synthesephase) andererseits vorsieht. Das Tycoon-System als persistente Programmierumgebung besitzt allerdings ein um ein Laufzeitsystem erweitertes Backend, in dem wesentliche Eigenschaften, die nicht in einem üblichen Übersetzer vorhanden sind, unterstützt werden. Die Realisierung der folgenden Merkmale des Tycoon-Systems wird maßgeblich von der Funktionalität und den Entwurfsentscheidungen der Schichten mit den Schnittstellen TML (Tycoon Machine Language), TVM (Tycoon Virtual Machine) und TSP (Tycoon Store Protocol) begünstigt:

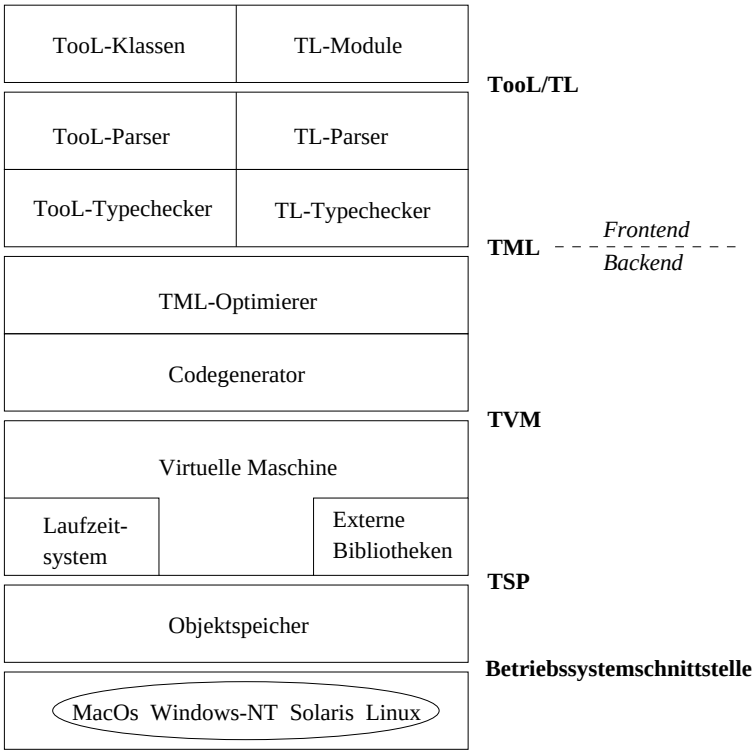


Abbildung 2.1: Architektur des Tycoon Systems

- Persistenz als Eigenschaft von beliebigen Werten (Daten, Funktionen, Threads usw.) orthogonal zum Typ
- Anwendungssprachen höherer Ordnung mit Funktionen als gleichberechtigte Werte
- Polymorphismus
- Offenheit für externe Bibliotheken durch typsichere Integration in die Hochsprache
- Portabilität von ausführbarem Code bis hin zur Migration von Threads

Diese Schichten werden in den folgenden Abschnitten kurz vorgestellt mit Erklärungen dazu, inwiefern sie zur Unterstützung dieser Sprach- und Systemeigenschaften beitragen.

2.1 TML: eine maschinenunabhängige Zwischensprache

Die Zwischensprache TML, eine Variante des Lambdakalküls im Fortsetzungsstil (*continuation passing style*), wird in Kapitel 5 detailliert erläutert. An dieser Stelle soll nur bemerkt werden, daß in TML wie in TL und TooL Funktionen Werte erster Klasse (*first class values*) sind. Das bedeutet, daß Funktionen gleichberechtigte Werte sind, die als Funktionsargumente und Funktionsergebnisse auftreten können. Programmiersprachen, in denen Funktionen gleichberechtigte Werte sind, werden als Sprachen *höherer Ordnung* bezeichnet. Somit

ist TML ebenfalls eine Sprache höherer Ordnung und unterstützt direkt diese Eigenschaft von TL und TooL. Eine maschinenunabhängige Optimierung findet auf der TML-Repräsentation statt. Der Codegenerator erzeugt aus dem TML-Zwischencode den ausführbaren Bytecode einer virtuellen Maschine TVM (Tycoon Virtual Machine).

2.2 TVM: eine portable virtuelle Maschine

Als Schnittstelle nach oben definiert die virtuelle Maschine TVM den Bytecode der Maschinenbefehle und das Format für Codeobjekte (*closures*, Ausführungsrepräsentation für Funktionsobjekte). Der Bytecode sieht neben allgemeinen Befehlen zur Ausführung von Programmen auch gezielte Mechanismen zur Unterstützung von Hochsprachenkonstrukten vor, z.B. einen `send`-Bytecode für den dynamischen Methodenaufruf in TooL. Der ausführbare Bytecode von Tycoon-Anwendungen ist portabel. Folgende Tycoon-Komponenten sind vollständig in TL geschrieben und liegen somit im portablen Bytecode-Format vor: TL- und TooL-Parser, Typprüfer für TooL und TL, Parsergenerator, Optimierer und Codegenerator.

Um einen TooL- oder TL-Übersetzer auf eine neue Hardwareplattform zu portieren, müssen lediglich das Tycoon-Laufzeitsystem, der TSP-Objektspeicher und der Bytecodeinterpreter angepaßt werden.

Die flexible Integration von Diensten externer Bibliotheken, die dynamisch zur Laufzeit in das System eingebunden werden, findet ihre Umsetzung in der TVM in Form eines speziellen `ccall`-Bytecodes. Argumente dieser TVM `ccall`-Operation sind der Name der Bibliothek und der C-Funktion, die dynamisch gebunden und aufgerufen werden soll. Diese `ccall`-Schnittstelle wird an den Parser der Hochsprache durch die TML-Schicht hindurch in Form eines ccall-TML-Primitivs gereicht.

Codeobjekte und Bytecode werden beim Übersetzen in den Tycoon Objektspeicher eingetragen, und auch Maschinenzustände des TVM-Interpreters sind TSP-Objekte. Da alle TSP-Objekte speicherbar sind, können laufende Threadzustände gespeichert oder über Netzwerkverbindungen versendet werden.

2.3 TSP: eine portable Objektspeicherschnittstelle

Jede Speicherung und Manipulation von Werten durch die Ausführung eines TVM-Programms findet im Tycoon Objektspeicher statt, der durch das Tycoon Store Protocol (TSP) definiert ist. Der Tycoon Objektspeicher bietet eine automatische Speicherverwaltung mit Speicherbereinigung (*garbage collection*) und eine sich den dynamischen Anforderungen anpassende Speichergröße. Alle Bindungen eines Objektspeicherzustandes bilden einen Objektgraphen, der von einem Wurzelobjekt aus erreichbar ist. Eine *commit*-Operation ermöglicht das Herausschreiben des gesamten Objektgraphen auf ein langlebiges Medium. Eine dazu komplementäre *restart*-Operation restauriert einen gespeicherten Zustand mit allen aktuellen Bindungen einer Sitzung. Fortgeschrittenere Sicherungs- und Rollbackmechanismen für das Tycoon-System sind in [Kornacker 95] beschrieben.

Die Implementierung des persistenten Objektspeichers mit automatischer Speicherverwaltung wird durch folgende Entwurfsentscheidung vereinfacht: alle Worte im Objektspeicher besitzen die gleiche Größe (32 Bits) und beschreiben ihren Typ durch zwei Tag-Bits.

Dadurch können während einer Speicherbereinigung Objektreferenzen von Ganzzahlen und Booleschen Konstanten auf einfache Weise unterschieden werden, und Objektdeskriptoren werden für Ganzzahlen und Wahrheitswerte unnötig. Ein weiterer Vorteil dieses Schemas ist die Unterstützung von Polymorphismus in der Hochsprache: da jedes Speicherobjekt durch ein Wort gleicher Größe identifiziert wird, sind alle Bindungs- und Zuweisungsoperationen auch auf Maschinenebene generisch. Der Code einer parametrisch polymorphen Funktion der Hochsprache kann auf die gleiche Weise Ganzzahlen, Wahrheitswerte oder komplexe Strukturen verarbeiten, denn die Objektreferenz einer komplexen Struktur unterscheidet sich nur durch die Tag-Kennung von einer Ganzzahl oder einem Booleschen Wert. Für eine arithmetische Bearbeitung der Ganzzahl ist keine Dereferenzierung nötig. Gleitkommazahlen werden dagegen als portable Bytefelder behandelt und müssen vor Operationen von dem Laufzeitsystem in das Format der jeweiligen Hardwareplattform konvertiert werden.

2.4 TooL im Tycoon System

Die Sprache TooL ist durch Austausch des TL-Frontends auf das bestehende Tycoon-Backend aufgesetzt, das, nur minimal verändert, weiterhin auch als TL-Plattform dient. Der TooL-Übersetzer ist im Gegensatz zu dem TL-Übersetzer nicht in der eigenen Quellsprache geschrieben, da dieses eine Re-Implementierung nicht nur des Frontends sondern auch des durch einen TL-*bootstrap* erprobten Backends bedeuten würde. Durch TooL werden die Möglichkeiten objektorientierter Modellierung mit der orthogonalen Persistenz des Tycoon-Systems vereint.

3. Die Programmiersprache TooL

Die Sprache TooL [Gawecki, Matthes 96] ist eine typisierte, rein objektorientierte Sprache mit Mehrfachvererbung und Metaklassen. Von vorrangigem Interesse in der vorliegenden Arbeit sind dabei die reine Objektorientierung und das Typsystem. Die reine Objektorientierung stellt eine der großen Hürden für den Optimierer dar. Einen Ansatzpunkt, die Kosten der reinen Objektorientierung zu verringern, liefert das Typsystem, allerdings unter Verlust der umfassenden Subsumption durch Subtypisierung. Nach einer kurzen Einführung in die Begriffe und Konzepte der objektorientierten Programmierung werden diese Aspekte der Sprache TooL im folgenden erläutert.

3.1 Objektorientierte Programmierung

Durch objektorientierte Programmiersprachen werden Mechanismen zur Verfügung gestellt, mit denen einige Probleme des Software-Engineerings behoben werden können: mangelnde Wartbarkeit von Software, fehlende Softwarebausteine[1] und ein Defizit an Modellierungsmächtigkeit. Objektorientierte Programmierung fördert die Wiederverwendung von Code und damit die Robustheit, denn im Idealfall müssen kritische Codefragmente nur einmal formuliert und auf Korrektheit überprüft werden. Durch Wiederverwendung können diese anschließend in verschiedenen Programmen zum Einsatz kommen. Erfolgreiche Wiederverwendung setzt voraus, daß genügend Codemuster, Datentypen und Algorithmen in kombinierbaren Softwarekomponenten vorliegen. Das wiederum bedeutet, daß eine Sprache das Herausfaktorisieren und die flexible Kombination von Codefragmenten unterstützen muß, eine Eigenschaft, die auch der Modellierung der Anwendungsdomäne zugute kommt. Die wesentlichen Eigenschaften objektorientierter Sprachen, die Wiederverwendung und Aggregation von Codefragmenten begünstigen, sind die Möglichkeit, Klassen zu definieren, Vererbung und der dynamische Methodenaufruf. Diese Eigenschaften werden in den nächsten Abschnitten erklärt.

3.2 Klassen und Vererbung

Objektorientierte Programmiersprachen erlauben die Strukturierung einiger oder aller Datentypen (je nach „Reinheit“ der Sprache) in einen gerichteten Graph von Klassen. Klassen

[1] [Cox 86] erwartet von Softwarebausteinen, daß sie im Sinne von elektronischen Bauelementen standardisiert und kombinierbar sein müssen; in seinem Buch wird daher der Begriff „Software-IC“ für einen Softwarebaustein geprägt.

kapseln Verhalten ähnlich wie abstrakte Datentypen und lassen Zugriffe nur über eine Schnittstelle an spezifizierten Operationen (*Methoden*) zu, wobei je nach Programmiersprache verschiedene Zugriffsmodalitäten und Einschränkungen möglich sind (z.B. *public* und *private* Methoden). Grundsätzlich muß eine Klassendefinition folgende Angaben beinhalten:

- Einen Klassennamen
- Eine oder mehrere Superklassen: Einige Programmiersprachen wie Smalltalk [Goldberg, Robson 83] lassen lediglich eine Superklasse zu. In anderen Sprachen, unter anderem auch in TooL, können Klassen mehrere Superklassen besitzen.
- Definition der Exemplarvariablen (*slots*): Exemplarvariablen stellen die Komponenten einer Klasse dar und tragen den Zustand eines Objekts. Je nach Sprache können diese Variablen über Methoden (z.B. in TooL oder Smalltalk) oder direkte Referenzen (z.B. in C++ [Ellis, Stroustrup 90]) angesprochen werden.
- Methodendefinitionen: In Methoden wird das dynamische Verhalten von Objekten einer Klasse beschrieben. Methoden entsprechen Prozeduren in imperativen Programmiersprachen.

Durch Angabe von Superklassen kann eine Klasse Funktionalität erben und diese durch Definition von neuen Methoden erweitern. Die über die Vererbungsbeziehung in Relation gebrachten Klassen bilden einen Klassengraphen, der in Systemen mit einfacher Vererbung (Klassen mit jeweils nur einer Superklasse) baumförmig und in Sprachen mit Mehrfachvererbung (Klassen mit mehreren möglichen Superklassen) als gerichteter Graph ausfällt. In Abbildung 3.1 ist ein Ausschnitt der TooL Massendatentyphierarchie dargestellt.

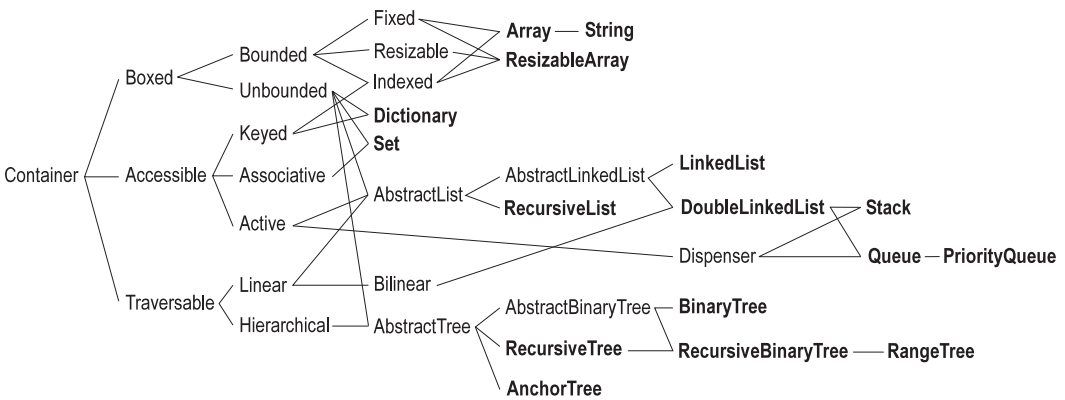
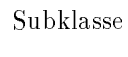


Abbildung 3.1: Die Massendatenhierachie in TooL (aus [Gawecki, Matthes 96])

Die folgenden TooL-Klassen liefern ein Beispiel für Vererbung (die Parametrisierung der Klassen mit einem Elementtyp E, der Typ Self sowie die Relation $<*$: werden in Abschnitt 3.5 eingeführt):

```
class Container(E <*: Equality)
super Object
public methods
isEmpty() :Bool
  ...
```




```
class Set(E <*: Equality)
super Container(E)
public methods
union(aSet :Self) :Self
  ...
```

In der Klasse Container sind Eigenschaften zusammengefaßt, die zu Behältern von Daten (Massendatentypen) gehören. Eine der Eigenschaften ist die Abfrage, ob ein Behälter leer ist (isEmpty). Eine Menge ist ein besonderer Behälter mit zusätzlichen Eigenschaften, daher wird die Klasse Set als *Subklasse* von Container definiert, unter anderem mit der Methode union. Die Methode isEmpty wird von der Klasse Container geerbt. In einer Subklasse können Methoden durch Implementationen überschrieben werden, in denen die konkrete Struktur der Subklasse berücksichtigt wird.

Objekte sind Exemplare von Klassen und werden in der Regel explizit erzeugt. Das Anlegen eines Objekts kann über ein Schlüsselwort der Sprache erfolgen (z.B. new in C++, create in Eiffel [Meyer 88]) oder über eine Operation eines Klassenobjekts, das wiederum ein Exemplar einer *Metaklasse* ist (z.B. in den Sprachen Smalltalk, CLOS [Bobrow et al. 88] und TooL).

3.3 Vererbung und der Aufruf von Methoden

Der Aufruf einer Methode eines Objekts wird aus historischen Gründen durch die von Smalltalk eingeführte Metapher einer Nachricht oder Botschaft (*message*) und eines Empfängers (*receiver*) beschrieben: ein Objekt empfängt eine Botschaft, die eine Operation des Objekts auslöst. Der Name einer Botschaft (z.B. „isEmpty“) wird *Selektor* genannt. Die Klasse des Empfängers einer Botschaft bestimmt, welche Implementierung einer Methode zur Laufzeit ausgewählt wird (dynamischer Methodenaufruf). Ein Methodenaufruf kann nicht statisch zur Übersetzungszeit an eine Methodenimplementierung gebunden werden, da eine Methode eines bestimmten Selektors in mehreren Klassen implementiert sein kann, und die konkrete Empfängerklasse nicht bekannt ist. Sogar Botschaften an das jeweilige Objekt selbst können nicht statisch gebunden werden, denn Methoden können in Subklassen überschrieben werden. Ein Beispiel verdeutlicht diesen Zusammenhang:

```
class A                                    class B
public                                     super A
methods                                    public
printHeader() { printDate.printId. ... }   methods
printId() { "A".print }                    printId() { "B".print }
  ...                                        ...
```

In TooL-Methoden erfolgen Botschaften an das jeweilige Objekt selbst durch das Schlüsselwort self, wobei die Angabe von self weggelassen werden kann. So ist der Aufruf printId in der Methode printHeader eine Kurzform für self.printId. Dieser Aufruf in der Methode printHeader, die von der Klasse B geerbt wird, kann je nach Empfängerklasse zur Laufzeit an die Implementierung printId(){"A".print} oder printId(){"B".print} gebunden werden.

Die prinzipiellen Schritte eines dynamischen Methodenaufrufs können wie folgt beschrieben werden:

1. Die Klasse des Empfängerobjekts wird bestimmt.
2. Wenn die Methode mit dem Selektor des Aufrufs in dieser Klasse definiert ist, kann sie aufgerufen werden.
3. Ist die Methode nicht in dieser Klasse implementiert, muß die Suche nach der passenden Methodenimplementierung in den direkten Superklassen mit Punkt 2 fortgesetzt werden, bis die Wurzelklasse erreicht wird. Sollte auch in der Wurzelklasse keine passende Methode definiert sein, tritt ein Laufzeitfehler auf.

Das auf diese Weise umschriebene Verfahren soll nur als Modell dienen, denn ein derart umgesetzter Methodenaufruf wäre nicht effizient genug für reale Implementierungen. Einen Überblick über gängige Techniken des dynamischen Methodenaufrufs, die in verschiedenen Sprachen zur Anwendung kommen, gibt [Driesen 93].

Der Vererbungsmechanismus ist in Verbindung mit dem dynamischen Methodenaufruf der Schlüssel zur Ausdrucksmächtigkeit objektorientierter Sprachen. Klassen können als Softwarebausteine verstanden werden, in denen Funktionalität zusammengefaßt ist. Die Kombination dieser Bausteine erfolgt über die klar definierten Schnittstellen der exportierten Methoden einer Klasse. Codemuster und Algorithmen werden nur einmal definiert und können über Vererbung wiederverwendet werden. Diese Vorteile tragen allerdings auch Kosten: die Effizienz von objektorientierten Programmen kann bis zu einer Größenordnung schlechter als die von imperativen Programmen sein. Zusätzlich schränkt der dynamische Methodenaufruf die Optimierungsmöglichkeiten stark ein. Als besonders ineffizient gelten rein objektorientierte Sprachen, deren Merkmale im nächsten Abschnitt beschrieben werden.

3.4 Reine Objektorientierung

Die Sprache TooL baut auf wenigen Grundmechanismen auf: dem dynamischen Methodenaufruf als einzigen Ausführungsmechanismus, dem Prinzip der Vererbung von Methoden an Subklassen und der Sicht von Funktionen als Objekte erster Klasse. Programmiersprachen, die nur den Methodenaufruf als Ausführungsmechanismus kennen, gelten als *rein objektorientiert*. TooL besitzt nur den Methodenaufruf als Sprachprimitiv für Ausdrücke, Kontrollflußsteuerung und Funktionsaufrufe. Es existieren keine speziellen syntaktischen Symbole in der abstrakten Syntax und keine gesonderte Codeerzeugung für Konditionale oder Schleifenkonstrukte. Diese Kontrollstrukturen werden über Botschaften modelliert und in der Sprache TooL selbst definiert.

Zur Verdeutlichung wird die Implementierung von if-then-else herangezogen. Im folgenden sind Ausschnitte der Klassen True und False abgebildet mit dem Code für die Methode „?:“, welche die Semantik von if-then-else implementiert:

```
class True
  "?:"(T <: Void, ifTrue, ifFalse :Fun():T):T
    {ifTrue[]}

class False
  "?:"(T <: Void, ifTrue, ifFalse :Fun():T):T
    {ifFalse[]}
```

Die Argumente dieser Methode sind zwei Funktionsobjekte (Blöcke), von denen in Abhängigkeit der Empfängerklasse nur eines ausgewertet wird. Empfängt ein Objekt der Klasse True die Nachricht „?:“, so wird das erste Argument ausgewertet (ifTrue). Wenn ein Objekt der Klasse False die Nachricht „?:“ empfängt, wird das zweite Argument (ifFalse) ausgewertet.

Als weiteres Beispiel soll die Methode while der Klasse Object dienen:

```
while(cond :Fun():Bool, statement :Fun():Void):Void
  { cond[]
   ? { statement[],
       while(cond, statement)
     }
  }
```

Die Kontrollstruktur while ist eine Methode der obersten Klasse Object im Klassengraphen. Da alle anderen Klassen von Object erben, ist sie in jeder anderen Methode als self.while aufrufbar. Da der Empfänger bei Methoden an self entfallen kann, lassen sich Schleifen wie von anderen Programmiersprachen gewohnt formulieren. Die „?“-Methode der Klassen True und False ist ein weiteres Konditional mit der Semantik von if-then und erwartet ein Funktionsobjekt als Argument. Für das Anlegen von Funktionsobjekten bietet TooL zwei syntaktische Varianten an: parameterlose Funktionen lassen sich durch geschweifte Klammern „{ }“, Funktionen mit Argumenten durch das Schlüsselwort fun(arg1 arg2 $\cdots$) erzeugen.

In der Implementierung von while wird eine parameterlose Funktion als Argument von „?“ für jeden Schleifendurchlauf angelegt. Hier wird ein schwerwiegender Nachteil der reinen Objektorientierung sichtbar: der Overhead einer while-Schleife ist beträchtlich. Statt eines einfachen Sprungs pro Schleifendurchlauf wird ein Funktionsobjekt erzeugt, und der Sprung erfolgt durch den rekursiven dynamischen Aufruf an die Methode while.

Diesem Nachteil steht der Vorteil einer möglichst einfachen und konzeptionell sauberen Definition des Sprachkerns mit einem kleinen Satz von Primitiven gegenüber. Es gibt keinen Unterschied zwischen vordefinierten und benutzerdefinierten Kontrollstrukturen. Zusätzlich verleiten keine gesondert codierten Konstrukte dazu, den objektorientierten Programmierstil aus Effizienzgründen zu übergehen. In Smalltalk sind Grundkontrollstrukturen fest codiert und deren Definition im Klassengraph dient lediglich der Dokumentation. Diese speziell codierten Kontrollstrukturen können dazu führen, daß abstrakte benutzerdefinierte Iteratoren wie select, reject usw. zugunsten der wenigen effizienten (imperativen) Kontrollstrukturen vernachlässigt werden, wodurch Programme an Generizität verlieren. Daher muß eines der

Hauptziele des TooL-Optimierers darin liegen, die Laufzeiteinbußen der elementaren Kontrollstrukturen und benutzerdefinierten Iteratoren auf gleiche Weise zu eliminieren und damit den rein objektorientierten Ansatz zu unterstützen.

3.5 Das TooL Typsystem

TooL ist eine streng typisierte Sprache mit Typinferenz, parametrischem Polymorphismus und Subtyppolymorphismus. In diesem Abschnitt wird erläutert, wie Typen von Klassen gebildet werden. Der Begriff eines Typs in TooL unterscheidet sich wesentlich von dem Begriff eines Typs in vielen anderen objektorientierten Sprachen. In Sprachen wie C++ und TS (*Typed Smalltalk*) [Johnson et al. 88] werden Typen mit Klassen gleichgesetzt. Die Sprache SELF [Ungar, Smith 87] sieht keine Klassen vor; stattdessen werden in dem Übersetzer klassenähnliche Strukturen (*maps*) verwaltet, die in den SELF-Übersetzern zu Optimierungszwecken als Typen betrachtet werden. In der Terminologie dieser Arbeiten spiegelt sich diese Gleichbehandlung wider: *type prediction* für Klassenabschätzung, *type analysis* für Klassenanalyse usw. In TooL sind Typen nicht Klassen, sondern Signaturen der abstrakten Datentypen, die durch Klassen realisiert sind. Mehrere Klassen können äquivalente Typen besitzen.

3.5.1 Strukturelle Typisierung

Im Gegensatz zur Sichtweise eines Typnamens als Identifikator eines abstrakten Datentyps (wie unter anderem in C, C++, und Modula-2) wird in TooL der Typ lediglich durch die Struktur der Datenobjekte beschrieben. Ein Datentyp ist demnach eine Menge von Signaturen und die Äquivalenz zweier Typen wird aufgrund von Struktureigenschaften festgestellt.

Der Typ eines TooL-Objekts ist eine Menge von Methodensignaturen (Funktionssignaturen) der Form $Methodenname : Fun(: A_1 \ldots : A_n) : B$, wobei $: Fun(: A_1 \ldots : A_n) : B$ den Typ einer Funktion mit n Argumenten der Typen $A_1 \ldots A_n$ und dem Ergebnistyp B beschreibt.

Typen können wie Klassen in einer Relation stehen, die allerdings von der Vererbungsrelation zu unterscheiden ist: In einer Typrelation kommt eine Beziehung zwischen den Signaturen von abstrakten Datentypen zum Ausdruck, während die Vererbungsrelation eine Beziehung zwischen Implementierungen ausdrückt. TooL unterstützt zwei Arten von Typrelation: die Subtyprelation und die Ähnlichkeitsrelation.

Bevor diese Relationen eingeführt werden, muß ein entscheidendes Konzept des TooL-Typsystems erläutert werden: Self.

3.5.2 Der Typ Self

Vererbung zwischen Klassen bedeutet, daß Methodensignaturen für verschiedene Typen gültig sein können. Wenn in dem folgenden Beispiel eine copyYourself-Methode geerbt wird, muß der Tatsache Rechnung getragen werden, daß sie je nach Empfängerklasse verschiedene Ergebnistypen liefern kann:

```
class A
public methods
copyYourself() :Self      ; liefert ein Objekt vom Typ der Klasse A
...

class B
super A
...                       ; erbt copyYourself, das aber jetzt Objekte vom Typ der Klasse B liefert
```

Hierfür existiert der spezielle Typ Self, der eine schwebende Selbstreferenz zum Ausdruck bringt und immer an den Typ der umgebenden Klasse gebunden wird. In dem Beispiel bedeutet der Ergebnistyp Self von copyYourself, daß diese Methode ein Objekt des Typs A zurückgibt, wenn ein Objekt der Klasse A der Empfänger der Methode ist. Ist dagegen ein Exemplar von B Empfänger der Methode, wird ein Objekt des Typs B zurückgegeben. Eine ausführliche Herleitung der Notwendigkeit eines Self-Typs ist in [Bruce 96] zu finden.

3.5.3 Subtypisierung

Die Substituierbarkeit von Objekten in Verwendungskontexten, in denen eigentlich Objekte eines anderen Typs erwartet werden, führt zum Begriff der Subtypisierung: Ein Typ A ist ein *Subtyp* ($<:$) eines anderen Typs B, wenn ein Objekt des Typs A statt des erwarteten Typs B verwendet werden kann. Daher subsumiert der Typ B den Typ A und auch alle weiteren Subtypen von B.

Wann die Substituierbarkeit von TooL-Objekten gilt, wird durch folgende Subtypregel bestimmt:

Der Objekttyp $\{a_i : A_i\}_{i\in[0\ldots n]}$ ist ein Subtyp eines anderen Objekttyps $\{b_j : B_j\}_{j\in[0\ldots m]}$, wenn für jede Signatur $b_j : B_j$ eine korrespondierende Signatur $a_i : A_i$ mit gleichem Methodennamen existiert, und A_i Subtyp von B_j ist:

$$\begin{aligned} \{a_i : A_i\}_{i\in[0\ldots n]} \quad <: \quad & \{b_j : B_j\}_{j\in[0\ldots m]} \\ \iff \quad & \forall\, b_j : B_j \in \{b_k : B_k\}_{k\in[0\ldots m]} \\ & \exists\, a_i : A_i \in \{a_l : A_l\}_{l\in[0\ldots n]} \\ & \Longrightarrow a_i = b_j \wedge A_i <: B_j \end{aligned}$$

Mit anderen Worten: Es gilt $A <: B$, wenn Methoden gleichen Namens in Subtypbeziehung stehen, und A eventuell mehr Methoden als B besitzt. Die Subtypregel der einzelnen Methoden ist dabei gemäß der Funktionssubtypisierung folgendermaßen:

Sei $Fun(:A) : B$ der Typ einer Funktion f, die Parameter vom Typ A übernimmt und Resultate vom Typ B berechnet. Um eine Funktion g vom Typ $Fun(:C) : D$ in dem Aufrufkontext von f zu substituieren, darf sie auf keinen Fall speziellere Typen als A erwarten, und ihr Ergebnistyp muß mindestens so speziell wie B sein. Daher gilt:

$$\begin{aligned} Fun(:C) : D \quad <: \quad & Fun(:A) : B \\ \iff \quad & A <: C \text{ und } D <: B. \end{aligned}$$

Bei der Funktionssubtypisierung wird von einer *kontravarianten* Beziehung zwischen den Argumenttypen gesprochen, weil diese gegenläufig zur Beziehung der beiden Funktionstypen ist. Die Relation zwischen den Ergebnistypen verläuft dagegen *kovariant* zur Beziehung der Funktionstypen.

An eine Methode kann jederzeit ein Subtyp des erwarteten Parametertyps übergeben werden, denn der Subtyp unterstützt alle erwarteten Operationen. Daß eventuell noch mehr Funktionalität unterstützt wird, ist nicht relevant, denn in dem Methodenrumpf wird nur auf die erwarteten Operationen zugegriffen. Die Möglichkeit, Methoden auf verschiedene Argumenttypen anzuwenden, sofern diese Subtypen der Formalparameter sind, heißt *Subtyppolymorphismus*. Auf den Subtyppolymorphismus wird in Abschnitt 6.3 noch eingegangen, denn das in der vorliegenden Arbeit entwickelte Optimierungsverfahren schränkt den Subtyppolymorphismus für einige Basisklassen ein.

Im allgemeinen stehen Klassen, die über Vererbung in einer Subklassenbeziehung stehen, nicht in einer Subtypbeziehung, wie die Ausschnitte der Klassen Point und ColoredPoint zeigen:

```
class Point                 class ColoredPoint
public                      super Point
x,y :Int                    public
methods                     color :String
add(aPoint :Self) :Self     methods
...                         isRed():Bool
                            ...
```

In diesem Beispiel erbt die Klasse ColoredPoint die Methode add der Klasse Point. Wenn ColoredPoint ein Subtyp von Point wäre, müßte ColoredPoint mindestens genau so viele Methoden wie Point besitzen, und für jede der gemeinsamen Methoden müßte die in ColoredPoint vertretene Methode Subtyp der in Point vertretenen Methode sein. Diese Forderung wird von der binären Methode add (als *binäre Methoden* werden Methoden mit einem Argument gleichen Typs bezeichnet) verletzt, denn für ColoredPoint nimmt der Typparameter Self den Typen ColoredPoint an und tritt daher in der Signatur von add als speziellerer Typ von Point kovariant statt kontravariant auf.

Um die Typbeziehung zwischen Subklassen einzufangen, bedarf es einer weiteren Relation, der sogenannten Ähnlichkeitsrelation.

3.5.4 Die Ähnlichkeitsrelation zwischen Objekttypen

Die Ähnlichkeitsrelation erlaubt die typkorrekte Definition von Subklassen, in denen Methodensätze mit binären Methoden geerbt, erweitert und überschrieben werden.

An dem Beispiel der Klasse ColoredPoint des vorherigen Abschnittes ist erkennbar, daß die flexible Bindung des Typs Self an den jeweiligen Umgebungstyp die Subtyprelation zwischen ColoredPoint und Point verhindert. Für die Ähnlichkeitsrelation (*type matching relation* [Abadi, Cardelli 95]) wird der Typ Self auf andere Weise interpretiert: ein Objekttyp A steht in der Ähnlichkeitsrelation zu einem Objekttyp B ($A <\!*\!: B$), wenn A Subtyp von B ist unter der Annahme, daß Self ein beliebiger fester Typ ist. Eine intuitive Vorstellung der Ähnlichkeitsrelation ist die folgende: Es gilt $A <\!*\!: B$, wenn die Klasse A mindestens die Methoden unterstützt, die die Klasse B besitzt. Nach dieser Definition paßt ColoredPoint zu Point, oder in Kurzform: ColoredPoint $<\!*\!:$ Point.

3.5.5 Subtypbegrenzung und Ähnlichkeitsbegrenzung

Die Vererbungseigenschaften einer Klasse lassen sich durch die Angabe eines Begrenzungstyps für Self bei der Klassendefinition wie folgt steuern:

- Eine Ähnlichkeitsbegrenzung im Stil von „Self <*: class A ...“ stellt den gängigsten Fall dar, der eine flexible Subklassenbildung ermöglicht. Für diesen Begrenzungsmodus existiert eine Kurzform „class A ...“ .
- Eine Subtypbegrenzung „Self <: class A ...“ schränkt die Subklassenbildung auf Klassen ein, die Subtypen des Typs A sind.
- Eine Äquivalenzbeschränkung „Self = class A ...“ verbietet die Spezialisierung von Methoden in Subklassen und die Definition neuer Methoden.

Die Wahl des Begrenzungsmodus hängt von der gewünschten Modellierung ab: Klassen, die höher im Vererbungsbaum liegen, werden eher mit einer Ähnlichkeitsbegrenzung definiert, um die Subklassenbildung möglichst wenig einzuschränken. Klassen, deren Methoden nicht überschrieben werden dürfen, können durch die Identitätsbegrenzung festgelegt werden. Eine Subtypbegrenzung erhöht die Subsumption für ganze Teilbäume im Vererbungsgraphen unter der Bedingung, daß Methoden mit Self-Parametern nicht spezialisiert und keine neuen Methoden angelegt werden können, wenn Self-Parameter existieren. Beispiele für die verschiedenen Begrenzungsmodi werden in [Gawecki, Matthes 96] ausführlich diskutiert.

3.5.6 Typparameter

TooL unterstützt neben dem Subtyppolymorphismus auch den begrenzten parametrischen Polymorphismus (*bounded parametric polymorphism*).

Klassen und auch einzelne Methoden können durch Typparameter an Generizität gewinnen und trotzdem typsicher übersetzt und angewendet werden. In der umfangreichen TooL Klassenbibliothek [Römer, Lotter 96] lassen sich vor allem Behälterklassen und Iterationsabstraktionen für Massendatentypen mit Hilfe von Typparametern generisch definieren. Die Extension der Parametrisierung ist durch Subtypbegrenzung oder Ähnlichkeitsbegrenzung der Typparameter regelbar.

Der folgende Ausschnitt der Klasse Set zeigt ein Beispiel für Klassen- und Methodenparametrisierung:

```
class Set(E <*: Equality)
public methods
add(e :E) :Void
includes(e :E) :Bool
map(F <*: Equality, f :Fun(:E):F) :Set(F)
...
```

Die Ähnlichkeitsbegrenzung des Typparameters E der Klasse Set gibt an, daß Elemente der Sets auf Subklassen von Equality beschränkt sind und somit immer eine Methode zum Gleichheitstest besitzen müssen. Die Methode map ist durch den Typparameter F derart parametrisiert, daß die Ergebnismenge der map-Iteration durchaus Elemente eines anderen Typs als die Ursprungsmenge haben darf.

4. Ansätze zur Optimierung objektorientierter Sprachen

Das Abstraktionsmodell in objektorientierten Programmiersprachen ist eine Strukturierung der Datentypen in eine Klassenhierarchie und der damit verbundene Vererbungsmechanismus. Methoden werden dynamisch zur Laufzeit gebunden, da erst durch die Empfängerklasse bestimmt werden kann, welche Implementierung einer Botschaft zum Empfängerobjekt paßt. Dieser Aspekt von objektorientierten Programmiersprachen ist eine *Abstraktionshürde* für optimierende Übersetzer, denn zur Übersetzungszeit ist nur die Signatur des durch eine Klasse realisierten abstrakten Datentyps bekannt. Klassen- oder Modulübergreifende Optimierungen sind zur Übersetzungszeit ausgeschlossen. Referenzen über Abstraktionshürden hinweg können in objektorientierten Sprachen erst während der Laufzeit aufgelöst und müssen immer wieder neu geknüpft werden, da die Empfängerklasse einer Botschaft sich von Aufruf zu Aufruf ändern kann. Dieser Vorgang verläuft naiv implementiert erheblich langsamer als ein statisch gebundener Funktionsaufruf, da sämtliche Oberklassen nach der passenden Implementierung durchsucht werden müssen. Selbst in der Sprache C++, in der ein lauffähiges Klassensystem einmal vor der Laufzeit gebunden wird, erfordern Methodenaufrufe in vielen Implementierungen drei Indirektionen zum Auffinden der passenden Methodenimplementierung. In einer rein objektorientierten Sprache wie TooL, in der in fast jedem Ausführungsschritt ein Methodenaufruf stattfindet, bedeutet der Aufwand des dynamischen Methodenaufrufs einen erheblichen Effizienzverlust. Neben diesem direkten Nachteil zeigt sich ein indirekter negativer Effekt: die vielen Methodenaufrufe an ungewisse Ziele erschweren Optimierungen oder schließen diese gänzlich aus.

Sind die Ziele von Methodenaufrufen allerdings eindeutig auflösbar, stehen funktionsübergreifenden Optimierungen ausgelöst durch Funktionsexpansion (*inlining*) nichts im Wege. Allen Strategien zur Auflösung von Referenzen ist gemeinsam, daß versucht wird, Information über Klassen von Ausdrücken zu gewinnen. In den folgenden Abschnitten werden Ansätze zur Gewinnung von Klasseninformation beschrieben, die zum einen Einfluß auf die vorliegende Arbeit haben, und zum anderen mit den Techniken des TooL-Optimierers verglichen werden sollen.

4.1 Klassenabschätzung

Für einen Satz an Basisselektoren wie „+,*,-“ usw. sind Empfängerklassen mit einer sehr hohen Wahrscheinlichkeit abschätzbar. In [Krasner 83] wird festgestellt, daß in Smalltalk 90%

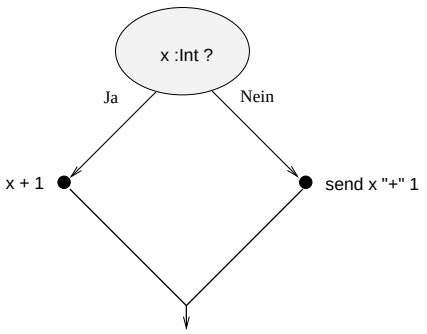


Abbildung 4.1: Einfache Klassenabschätzung

aller Aufrufe von Methoden mit dem Selektor „+“ an Objekte der Klasse SmallInteger gerichtet sind. Ähnliches gilt für andere arithmetische und logische Selektoren; auch hier sind die Empfängerobjekte mit sehr hoher Wahrscheinlichkeit Exemplare von SmallInteger bzw. Boolean. Während der Ausführung von Methodenaufrufen mit diesen speziell gekennzeichneten Selektoren kann ein Klassentest erfolgen. Ist dieser erfolgreich, wird eine primitive Operation direkt ohne Methodenaufruf ausgeführt; im anderen Fall einer unerwarteten Empfängerklasse ist ein regulärer Methodenaufruf erforderlich. Dieses Verfahren heißt Klassenabschätzung (*type prediction*) und ist in Abbildung 4.1 angedeutet.

Durch Klassenabschätzung können die Kosten des dynamischen Methodenaufrufs für einige Basisselektoren bei erfolgreicher Klassenvorhersage auf den wesentlich geringeren Aufwand einer Vergleichsoperation und eines bedingten Sprungs reduziert werden.

Schon in dem mikrocodierten Interpreter für Smalltalk-76 [Ingalls 78] wird eine Klassenabschätzung bei der Interpretation von Basisselektoren durchgeführt. In dem Smalltalk-Übersetzer für konventionelle Rechnerarchitekturen von Deutsch und Schiffman [Deutsch, Schiffman 84] werden bei der Übersetzung des Smalltalk-Bytecodes in Maschinencode Methodenaufrufe durch Klassenabschätzung vermieden. Mittlerweile gehört diese Technik zu einer vielfach eingesetzten Optimierung für objektorientierte Sprachen, z.B. in SELF [Ungar, Smith 87] und Smalltalk [Deutsch, Schiffman 84; Ungar 87; Gawecki 92].

4.2 Inline Caching

Einige Smalltalk-Übersetzer verwenden eine zentrale Hashtabelle als Methodencache, in den ein Zeiger auf den aufgefundenen Code eines Methodenaufrufes eingetragen wird. Der Hashindex wird durch eine sehr einfache Funktion aus dem Methodenselektor und der Empfängerklasse berechnet, z.B. durch Addition des Selektors mit dem Klassenindex modulo der Tabellengröße. Für erneute Aufrufe einer gegebenen Kombination von Selektor und Empfängerklasse wird der Aufwand des dynamischen Methodenaufrufs im Idealfall auf die Berechnung des Hashindex und Schlüsselvergleichs verringert. Ein Vergleich von Hashfunktionen für das Caching von Methoden in Smalltalk-Systemen ist in [Krasner 83] zu finden.

Ausgehend von der Feststellung, daß Empfängerklassen an Aufrufpositionen weitgehend konstant bleiben, wurde in dem Deutsch-Schiffman-System der Methodencache in den

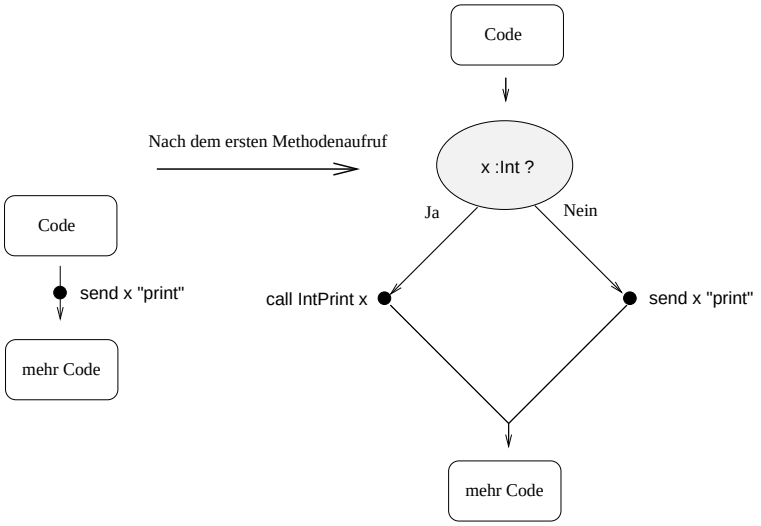


Abbildung 4.2: Prinzip des Inline-Caching

ausführbaren Code integriert (*inline caching*): Ein zur Laufzeit auftretender Methodenaufruf wird nach dem dynamischen Auffinden der passenden Methode an der Aufrufstelle ersetzt durch eine Klassenüberprüfung und einen Sprung an den aufgefundenen Code. Die Integration des Methodencaches in den ausführbaren Code ersetzt den Aufwand der Hashverwaltung durch einen einfachen Klassentest. Diese Idee ist in Abbildung 4.2 skizziert. Auch das Inline-Caching wird in verschiedenen Systemen eingesetzt, z.B. in dem SELF-Übersetzer von Chambers [Chambers 92] oder dem Smalltalk-Übersetzer für einen dedizierten Smalltalk-RISC-Prozessor SOAR [Ungar et al. 84].

Um Aufrufe mit wechselnden Empfängerklassen ähnlich effizient zu gestalten, wurde in [Hölzle 94] der Inline-Cache zu einem polymorphen Inline-Cache weiterentwickelt; der Grundgedanke ist in Abbildung 4.3 dargestellt. Für eine bestimmte Zahl von Empfängerklassen werden Klassentests in den Code eingefügt, und somit lassen Methodenaufruforte mit variierenden Empfängerklassen trotz Polymorphismus optimierte Methodenaufrufe zu. Zusätzlich bildet der integrierte Cache in SELF eine wichtige Informationsquelle, die für dynamische Optimierungen zur Laufzeit ausgenutzt wird. Ein polymorpher Inline-Cache stellt zusammen mit einem Methodenaufrufzähler ein Laufzeitprofil dar, das zwei Arten von Information an einen Optimierer liefert: Erstens zeigt ein integrierter Cache an, an welchen Stellen eine Methodenintegration möglich wird (überall, wo Aufrufe durch einen Prozeduraufruf realisiert sind), und zweitens werden Hinweise darüber gegeben, wo eine Methodenintegration besonders effektiv werden könnte (Aufrufpositionen mit einer hohen Aufrufdichte).

4.3 Klassenanalyse und Splitting

Ausgehend von Hinweisen oder konkreten Informationen über die Klassen einiger Empfänger kann ein Analyselauf weitere Empfängerklassen offenlegen, etwa durch Inferenzen über Ergebnisklassen von primitiven Operationen. In [Johnson et al. 88] und [Atkin-

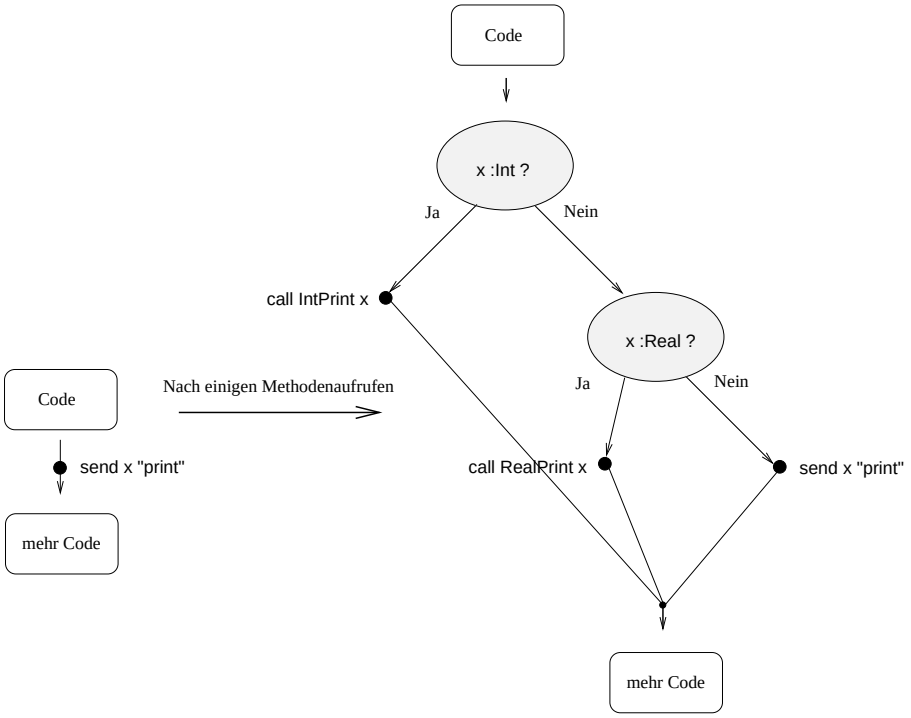


Abbildung 4.3: Polymorphes Inline-Caching

son 86] werden typisierte Varianten der Sprache Smalltalk vorgestellt, in denen explizit im Programmtext angegebene Klasseninformation in einer Klassenanalyse für weiterführende Optimierungen ausgenutzt wird, die in den Übersetzern statisch vor der Laufzeit erfolgen. Auch durch Klassenabschätzung und Inline-Caching wird Code um Klasseninformation bereichert, die durch eine Klassenanalyse weitere Optimierungen zuläßt. Der SELF-Übersetzer von Chambers [Chambers 92] führt eine Klassenanalyse durch, die einer Datenflußanalyse ähnelt: Klasseninformation in der Form von Mengen von möglichen Objektklassen wird auf Kontrollflußpfaden durch einen Kontrollflußgraphen propagiert. Durch Verschiebung von Code über Vereinigungsknoten des Kontrollflußgraphen hinaus (*splitting*) werden monomorphe Strecken vergrößert und wiederholte Klassentests vermieden, wie das Beispiel in Abbildung 4.4 andeutet. Aufgrund der Vielzahl von Klassentests, die durch einfache Klassenabschätzung und Inline-Caching im Code eingeführt werden, ist die Verschiebung von Code über Knoten hinaus eine wirkungsvolle (und nötige) Optimierung in den SELF-Übersetzern.

4.4 Profilgesteuerte Optimierung

Information aus vergangenen Programmläufen kann Aufschluß über allgemeines Ablaufverhalten und zukünftige Programmläufe geben, sofern der Beobachtungszeitraum eine repräsentative Stichprobe zuläßt. Eine gute Stichprobe kann sich aus dem Profiling von Anwendungsprogrammen ergeben, denn so gewonnene Daten entsprechen eher den realen Anforderungen an ein Softwaresystem als Messungen an Testprogrammen. Die Frage der Gültigkeit und der

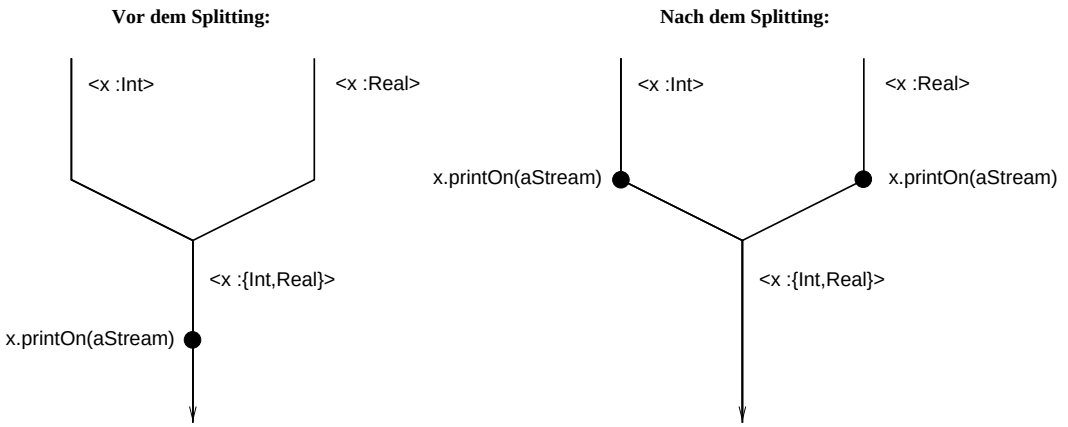


Abbildung 4.4: Splitting

Granularität von Profildaten für die objektorientierten Sprachen Cecil [Chambers 93] und C++ wird von [Grove et al. 95] untersucht. Zusätzlich wird die Wirksamkeit verschiedener Arten von Profilinformationen zur Klassenabschätzung gemessen. Grove et al. klassifizieren die Granularität von Profilinformation nach der Tiefe einer beobachteten Aufrufkette (*call chain*). Die gröbste Einheit stellen Aufrufhäufigkeiten von Methoden dar, die über alle Aufruforte hinweg gemittelt werden (z.B. realisiert durch einen Aufrufzähler für jede Methode). Folgen von aufrufortspezifischen Häufigkeiten geben genauer Aufschluß über Aufrufdichten. So trägt ein polymorpher Inline-Cache zusätzlich zu den Aufrufhäufigkeiten auch noch aufrufortspezifische Information und wäre damit die nächste Granularitätsstufe. Weitere Stufen sind Folgen von Aufruforten im Sinne von verketteten Aktivierungssegmenten eines Kellers.

Für Optimierungen durch Klassenabschätzungen erweisen sich einfache (nicht verkettete) aufrufortspezifische Informationen am nützlichsten, da diese eine gezielte Methodenintegration ermöglichen, gefolgt von den einfachen gemittelten Häufigkeiten. Die Gültigkeit von Profildaten bleibt nach den Messungen von [Grove et al. 95] über längere Zeiträume und über verschiedene Programmversionen hinweg erhalten, so daß Profiling ein wirkungsvolles Werkzeug zur Optimierung von objektorientierten Sprachen zu sein scheint.

Laut [Cutts et al. 94] stellen Programmiersysteme mit orthogonaler Persistenz und Reflektion wie das Tycoon-System eine geeignete Umgebung dar, um Profiling ohne größere Schwierigkeiten zu integrieren. In der Arbeit von Cutts et al. wird die polymorphe persistente Sprache Napier-88 [Morrison et al. 94] um die Möglichkeit, Code aufgrund von Information in Laufzeitprofilen erneut zu übersetzen, erweitert. Auf diese Weise werden mehrere Codeversionen einer Prozedur erzeugt, unter denen in Abhängigkeit von Laufzeitbedingungen ausgewählt wird. Die Entwicklung des Codebestands nimmt evolutionäre Züge an, denn nach einer Optimierung kann das Laufzeitverhalten weiter in einem Profil festgehalten werden. Nicht mehr benötigte Codesegmente (weil z.B. nur noch die optimierte Version aufgerufen wird) können aus dem System herausfallen.

4.5 Empfängerspezifische Übersetzung

Der Einsatz eines zentralen oder integrierten Methodencaches erlaubt die effektive Optimierungstechnik der empfängerspezifischen Übersetzung (*customization*) [Chambers, Ungar 89] [Gawecki 92]. Da nach einem Klassentest oder einer Hashberechnung die Empfängerklasse eines Methodenaufrufs bekannt ist, kann aus der aufgerufenen Methode eine auf die Empfängerklasse zugeschnittene Version erzeugt werden, die daraufhin in den Methodencache eingetragen wird. In dieser speziellen Version einer Methode lassen sich alle Methodenaufrufe an den Empfänger self durch statische Funktionsaufrufe oder expandierte Codefragmente (*inlining*) ersetzen, da die Klasse von self für diese Methodenversion bekannt ist.

Dem Nachteil der Redundanz der auf verschiedene Empfängerklassen zugeschnittenen Codeversionen stehen die erweiterten Optimierungsmöglichkeiten gegenüber. Gerade in rein objektorientierten Sprachen, in denen auch Exemplarvariablen eines Objekts über Botschaften an self referenziert werden, ist die empfängerspezifische Übersetzung eine effektive Optimierung. In Systemen mit dynamischer Übersetzung wird nur der aktuell benötigte Code im Cache gehalten und bei Bedarf erneut übersetzt, so daß der Speicherverbrauch der spezialisierten Methodenversionen begrenzt ist.

5. Lambdakalkül im Fortsetzungsstil als Zwischensprache

Aufgrund der großen semantischen Lücke zwischen Konstrukten von Hochsprachen und den ausführbaren Maschinenbefehlen werden beim Übersetzungsvorgang Transformationen über mehrere Zwischenrepräsentationen durchgeführt. Eine der Zwischenstufen dient als Repräsentation für Optimierungsalgorithmen, wobei diese je nach Quellsprache, Grad der erwünschten Maschinenunabhängigkeit und erhofftem Optimierungserfolg mehr oder weniger in der Mitte der semantischen Lücke angesiedelt sein kann. Eine häufig verwendete Zwischensprache für Übersetzer imperativer Sprachen ist der sogenannte „Drei-Adreß-Code", dessen einzelne Instruktionen aus Tupeln der Form <Operation, Zielvariable, Quellvariable1, Quellvariable2> bestehen. Die Vorzüge dieser Repräsentation für imperative Sprachen liegen in der einfachen Generierbarkeit aus dem abstrakten Syntaxbaum und in der Freilegung aller Registeranforderungen. Ferner dient sie als Grundlage vieler Optimierungs- und Datenfluß-algorithmen, die in [Aho et al. 86] beschrieben sind. Ein gutes Beispiel für einen optimierenden Übersetzer, der auf Drei-Adreß-Code Optimierungen durchführt, ist der PL.8-Übersetzer für verschiedene Quell- und Zielsprachen [Auslander, Hopkins 82].

Drei-Adreß-Code ist für das Tycoon-System aus folgenden Gründen allerdings weniger gut geeignet:

- Die semantische Ebene des klassischen Drei-Adreß-Codes ist, gemessen an den Anforderungen des Tycoon-Systems, zu nahe an realem imperativen Maschinencode. In der Zwischensprache sollten Funktionen gleichberechtigte Werte sein. Da TooL und TL Sprachen höherer Ordnung und von einer sehr hohen Funktionsaufrufdichte (oder Methodenaufrufdichte) gekennzeichnet sind, muß eine Zwischensprache gerade die Manipulation und Elimination von Funktionen und Funktionsaufrufen ermöglichen.
- Optimierungsalgorithmen für Drei-Adreß-Code setzen aufwendige Datenflußanalysen voraus und zielen in der Regel auf die Optimierung einzelner Funktionen ab. Der von TooL und TL geförderte Programmierstil führt zu vielen kurzen Funktionen, die von den Abstraktionshürden der modularen Programmierung voneinander getrennt sind und somit wenig Angriffsfläche für datenflußgesteuerte Optimierungen bieten. Die Hauptaufgabe eines Optimierers für TooL muß darin liegen, die Abstraktionshürden der Methoden- und Funktionsaufrufe zu überwinden.

Die erweiterte Variante des Lambdakalküls CPS (*Continuation Passing Style*), die sich als Zwischensprache in vielen Übersetzern, unter anderem für Scheme [Steele 78; Bartley, Jensen

86; Kranz et al. 86; Teodosiu 91], ML [Appel 89], Pascal und BASIC [Kelsey, Hudak 89], Smalltalk [Gawecki 92] und TL [Gawecki, Matthes 94; Kiradjiev 94], bewährt hat, erfüllt obige Kriterien und bietet sich als Optimierungsrepräsentation für TooL an, wie sich im folgenden Abschnitt zeigen wird.

5.1 TML

$lit \in Lit$		Literale, Konstanten und Objektidentifikatoren	
t ∈ Temp		Temporäre Variablen	
c ∈ Cont		Fortsetzungsvariablen	
v ∈ Var		Variablen (Bezeichner)	
prim ∈ Prim		Primitive Operationen	
val ∈ Val		Werte	
abs ∈ Abs		Abstraktionen	
app ∈ App		Applikationen	
v	::=	$t \mid c$	
val	::=	$lit \mid v \mid abs$	
abs	::=	$\lambda(v_1..v_n)\, app$	n ≥ 0
app	::=	$(val_0\, val_1..val_n)$	n ≥ 0
	|	$(prim\, val_1..val_n)$	n ≥ 0

Abbildung 5.1: Abstrakte Syntax der Zwischensprache TML (aus [Gawecki, Matthes 94])

Die Zwischensprache TML (Tycoon Machine Language), die den Kern des Tycoon-Backends stark prägt, besitzt folgende Eigenschaften:

- Die abstrakte Syntax von TML fällt einfach aus; in Abbildung 5.1 ist die gesamte Syntax aufgeführt. Wie im Lambdakalkül wird zwischen Applikationen und Abstraktionen unterschieden. Applikationen (*app*) sind Funktionsaufrufe, Anwendungen primitiver Operationen und reduzierbare Lambda-Terme. Abstraktionen (*abs*, Lambda-Terme in Normalform) sind Funktionsobjekte, die sich wie gewöhnliche Werte manipulieren lassen. Damit sind Funktionen in TML gleichberechtigte Werte.

- Funktionsaufrufe und Anwendungen primitiver Operationen unterliegen einer Fortsetzungssemantik. Jede Funktion und jede primitive Operation erwartet bis zu zwei zusätzliche Parameter, eine normale Fortsetzung und eine Ausnahmefortsetzung. Eine Fortsetzung (*continuation*) ist eine spezielle Funktion, die den Rest einer Berechnung beschreibt, wobei die normale Fortsetzung den Berechnungsverlauf ohne Auftritt eines Fehlers oder einer Ausnahme beinhaltet und die Ausnahmefortsetzung die Reaktion auf Ausnahmen oder Fehler festlegt. Fortsetzungen werden implizit nach einem Funktionsaufruf oder einer primitiven Operation aufgerufen und stellen im Gegensatz zu Abstraktionen keine Objekte erster Klasse dar. Dieses kommt dadurch zum Ausdruck,

daß Fortsetzungen niemals Fortsetzungen als Argumente übernehmen können, nicht gespeichert und nicht als Ergebnis zurückgegeben werden können.

- Der Rücksprung einer Funktion (return) an die Aufrufstelle erfolgt über eine Rücksprungfortsetzung.

- Neben Primitiven zur Manipulation und Erzeugung von Feldern und zur Berechnung von Werten existieren Operationen für die Steuerung des Kontrollflusses, zur Definition von rekursiven Funktionen und für den dynamischen Methodenaufruf. Einige der primitiven Operationen sind in Abbildung 5.2 zusammengefaßt.

- Die Auswertung erfolgt strikt und die Auswertungsreihenfolge liegt fest, da Applikationen keine Argumente sein können. Somit treten keine geschachtelten Aufrufe mehr auf, und alle Berechnungen sind linearisiert.

(send $sel\ r\ arg_1 \ldots arg_n\ c$)	Aufruf der Botschaft sel mit Empfänger r
(sendSuper $sel\ self\ arg_1 \ldots arg_n\ c$)	Aufruf der Botschaft sel an eine Superklasse
(== x	Konditional, beruhend auf Objektidentität, mit
$v_1 \ldots v_n$	Werten und
$c_1 \ldots c_n\ [c_{n+1}]$)	Verzweigungen (optionaler *else*-Zweig)
(classTest v	Konditional, beruhend auf
$class_1 \ldots class_n$	Klassenzugehörigkeit von v
$c_1 \ldots c_n\ [c_{n+1}]$)	
(Y $\lambda(c\ v_1 \ldots v_n)\ app$)	Y-Kombinator für rekursive Definitionen
($p\ x\ y\ c_e\ c_c$)	Ganzzahlige Arithmetik, $p \in \{+, -, *, /, \%\}$
($p\ x\ y\ c_1\ c_2$)	Vergleichsoperationen, $p \in \{<, >, <=, >=\}$
($p\ x\ y\ c$)	Bit-Operationen, $p \in \{<<, >>, \&, \mid, \hat{}, \tilde{}\}$
(array $val_1 \ldots val_n\ c$)	Anlegen eines veränderlichen Feldes
	mit n Einträgen
(classnew $class\ n\ c$)	Anlegen eines Exemplars der Klasse $class$ mit n slots
([ ] $x\ i\ c$)	Feldzugriff
([ ] := $x\ i\ v\ c$)	Feldzuweisung
(ccall $fmt\ cfn\ c_1\ c_2$)	C-Funktionsaufruf

Abbildung 5.2: Einige primitive TML Operationen (aus [Gawecki,Mattes 94])

In Textdarstellungen werden Abstraktionen durch das Schlüsselwort proc, Lambda-Applikationen durch lambda und Fortsetzungen durch cont gekennzeichnet. Am Beispiel des TML-Codes für die Fakultätfunktion lassen sich die Mechanismen des Fortsetzungsstils illustrieren:

```
(Y proc(c_1 fac)                ; Y-Kombinator zur Definition der rekursiven Funktion fac
 (c_1
   cont()
    (fac 10 cont(t_1)           ; Aufruf von fac mit 10
    (cc t_1))                   ; Ergebnis (Fakultät von 10) an die Aufrufumgebung
   proc(n ce c)                 ; Beginn des Rumpfes von fac
    (== n 1
       cont()
        (c 1)
       cont()
        (- n 1 ce cont(t_2)
        (fac t_2 ce cont(t_3)   ; rekursiver Aufruf von fac
        (* n t_3 ce cont(t_4)
        (c t_4)))))
```

Das Y-Primitiv (benannt nach dem Y-Kombinator des Lambdakalküls) dient der Definition von rekursiven Funktionen, die während der Codegenerierung einer gesonderten Behandlung unterliegen. Ein Y-Primitiv $(\mathrm{Y}\ \lambda(c\ v_1 \ldots v_n)\ (c\ cont() \ldots proc_1 \ldots proc_n)$ teilt dem Codegenerator mit, daß die Bezeichner $v_1 \ldots v_n$ an die möglicherweise wechselseitig rekursiven Abstraktionen $proc_1 \ldots proc_n$ gebunden werden, und daß die Berechnung mit $cont()$ fortgesetzt wird. Auf das Beispiel der Fakultätfunktion übertragen, bedeutet dieses: der Bezeichner fac wird an die rekursive Funktion proc(n ce c) (...) gebunden und der weitere Verlauf der Berechung wird durch cont() (fac 10 ...) beschrieben.

In diesem Beispiel besteht der weitere Verlauf der Berechnung aus dem Aufruf der durch das Y-Primitiv definierten Funktion fac mit 10. Das Ergebnis des Aufrufs wird an t_1 gebunden durch die Fortsetzungsapplikation cont(t_1)(...) und an die Rücksprung-Fortsetzung der Aufrufumgebung des Y-Primitivs (cc) in dem Term (cc t_1) übergeben.

Die zusätzlichen Fortsetzungsargumente c (Rücksprung-Fortsetzung für fac) und ce (Ausnahme-Behandlung für fac) werden in dem Funktionskopf proc(n ce c) neben dem Argument n der Fakultätfunktion aufgeführt.

Das folgende Konditional im Rumpf der Fakultätfunktion drückt den Sachverhalt „*wenn* **n** *eins ist, ist das Ergebnis eins, sonst* **n** *multipliziert mit Fakultät von* **n** *minus eins*“ aus:

```
(== n 1                         ; ist n gleich 1 ?
   cont()                       ;  - Ja,
    (c 1)                       ;    das Ergebnis ist 1
   cont()                       ;  - Nein,
    (- n 1 ce cont(t_2)         ;    t_2 ← n - 1
    (fac t_2 ce cont(t_3)       ;    t_3 ← fac(t_2)
    (* n t_3 ce cont(t_4)       ;    t_4  ← n * t_3
    (c t_4)))))                 ;    das Ergebnis ist t_4
```

Die Aufrufe (c 1) und (c t_4) der Rücksprungfortsetzung können als return-Operationen verstanden werden; im Vorgriff auf Kapitel 7 sei an dieser Stelle erwähnt, daß der Codegenerator dementsprechend `return`-Bytecodes für solche Applikationen erzeugt.

5.2 Die Transformation von TooL nach TML

Class	::=	class *identifier* public slots methods private slots methods
slots	::=	slots *identifier* | ϵ
methods	::=	methods *identifier*(parameters) bindings | ϵ
parameters	::=	parameters *identifier* | ϵ
bindings	::=	bindings *identifier*=value | bindings $\diamond$=value | ϵ
value	::=	const | *identifier* | self | super | fun(parameters) bindings | send(*identifier*, value, bindings)

Abbildung 5.3: Vereinfachte abstrakte Syntax von TooL

Da TooL eine rein objektorientierte Sprache ist, besitzt die abstrakte Syntax von TooL, abgesehen von den Produktionen des Typsystems, nur wenige Konstrukte, denn für Ausdrücke und für den Kontrollfluß existiert nur der Methodenaufruf (*message send*). Die wesentlichen Sprachelemente sind in einer vereinfachten abstrakten Syntax ohne Produktionen des Typsystems zusammengefaßt, die in Abbildung 5.3 dargestellt ist. Die Abbildung dieser Konstrukte soll im folgenden erläutert werden. Hierzu werden der TooL-Code, der abstrakte Syntaxbaum und der TML-Code der Methode while gegenübergestellt:

```
while(cond :Fun():Bool, statement :Fun():Void):Void
  { cond[]
    ? { statement[],
        while(cond, statement)
      }
  }
```

Aus dem TooL-Quelltext erzeugt der TooL-Parser einen abstrakten Syntaxbaum, in dem alle Bindungen und alle Methodenaufrufe explizit aufgeführt sind. Anonyme Bindungen ohne Bezeichner werden durch „$\diamond$=...“ dargestellt. Gemäß der Syntax in Abbildung 5.3 sieht dieser für while wie folgt aus:

```
while(cond,statement)
  ◇ = send(?,send([],cond,ϵ),
           ◇ = fun()
                 ◇ = send([],statement,ϵ)
                 ◇ = send(while,self,
                          ◇ = cond
                          ◇ = statement))
```

Die wesentlichen Punkte der Transformation des abstrakten Syntaxbaums in folgende TML-Repräsentation werden anschließend erläutert:

```
proc(self cond statement e c)
  (send "[]" cond e cont(t_1)
  (send "?" t_1
    proc(e_2 c_3)
     (send "[]" statement e_2 cont(t_4)
     (send "while" self cond statement e_2 cont(t_5)
     (c_3 t_5)))
    cont(t_6)
      (c t_6)))
```

1. Methodendefinitionen werden auf Abstraktionen (Funktionen) abgebildet, wobei die Liste der formalen Parameter um eine Rücksprungfortsetzung c, eine Ausnahmefortsetzung[1] e und einen Parameter self für das Objekt selbst erweitert wird. Die Notwendigkeit des self-Parameters ist durch mögliche Selbstreferenzen (Botschaften an self) in dem Rumpf der Methode gegeben. Auf diese Weise werden zum Beispiel Exemplarvariablen (*slots*) eines Objekts referenziert, denn ein Objekt besteht aus einer Referenz auf die Klasse und dem Feld der Exemplarvariablen. In dem Beispiel der while-Methode besteht der Rumpf aus einer anonymen Bindung.

2. Bindungen können auf drei Weisen nach TML transformiert werden:

   a) Bindungen an berechnete Werte (in TooL sind berechnete Werte Ergebnisse von send-Ausdrücken) erfolgen in TML durch den Aufruf von Fortsetzungen. Durch die Konversion in den Fortsetzungsstil werden geschachtelte Berechnungen linearisiert und temporäre Variablen eingeführt, die ebenfalls durch Fortsetzungen gebunden werden. In dem Code für while geschieht dies für das Ergebnis t_1 von (send "[]" cond ...).

   b) Bindungen an konstante Werte (Zahlen, Zeichenketten und Funktionen) werden durch anonyme Lambda-Applikationen durchgeführt. In dem while-Code treten derartige Bindungen nicht auf. Ein Beispiel ist:

   ```
   x = 3, ...   ⟶   (lambda(x)
                       ( ··· )
                     3)
   ```

   c) Von anonymen Bindungen in Argumentlisten eines send-Aufrufs werden in dem TML-Code nur Werte übernommen, zum Beispiel:

[1] Ein getrennter Lauf, der ausführlich in [Kiradjiev 94] beschrieben ist, wandelt vor der Codegenerierung Definitionen von Ausnahmefortsetzungen in Aufrufe an das TML-Primitiv pushHandler um. Beim Ende eines Blocks, für den eine Ausnahmefortsetzung gilt, wird eine popHandler-Anweisung eingefügt. Nach diesem Lauf treten Ausnahmefortsetzungen nur noch bei primitiven arithmetischen Operationen explizit auf. Aus diesem Grund werden Ausnahmefortsetzungen in den meisten Beispielen fortan weggelassen.

```
send(while,self,
     ◇ = cond       ⟶ (send "while" self cond statement e_2 ...)
     ◇ = statement))
```

3. Funktionen in TooL (im abstrakten Syntax-Baum sind diese durch das Schlüsselwort fun() gekennzeichnet) werden auf Abstraktionen in TML abgebildet. Die parameterlose Funktion, die im while-Beispiel an den Aufruf der ?-Methode übergeben wird, liegt in dem TML-Code als Abstraktion proc(e_2 c_3) (...) vor. Diese Abstraktion birgt durch den Mechanismus des Methodenaufrufs die Rekursion an while.

5.3 Reduktion als grundlegendes Optimierungprinzip von CPS

Die Reduktionsregeln des Lambdakalküls, die zur Auswertung von Lambda-Ausdrücken dienen, bilden das Grundprinzip des TML-Optimierers, der in [Kiradjiev 94] beschrieben ist. In diesen Regeln finden die Mechanismen der Bindung an Bezeichner, der Substitution von gebundenen Werten und der Applikation von Lambda-Termen eine Formalisierung.

5.3.1 Beta-Reduktion

In der folgenden Variante des Lambdakalküls bestehen Lambda-Terme aus dem Symbol λ, einem Parameter und dem Rumpf (der wiederum ein Lambda-Term ist), aus Applikationen von Lambda-Termen, oder einfach aus Bezeichnern:

$$\begin{aligned} \lambda\text{-Term} &:= \lambda(v)E_1 \mid (E_1\ E_2) \mid x \\ E &:= \lambda\text{-Term} \end{aligned}$$

Der Parameter v gilt in dem Ausdruck $\lambda(v)E_1$ als gebunden. Treten Bezeichner im Rumpf auf, die nicht als Parameter eingeführt wurden, heißen diese *frei*. Eine Applikation eines Lambda-Terms auf ein Argument bedeutet im Lambdakalkül, daß das Argument für alle Vorkommen des Parameters im Rumpf des Lambda-Terms substituiert werden und das äußere λ wegfällt (Beta-Reduktion):

$$(\lambda(v)E_1)E_2 \quad \xrightarrow{\beta} \quad E_1[E_2/v]$$

Der Ausdruck $E_1[E_2/v]$ steht für den Lambda-Term E_1, in dem alle Vorkommen von v durch E_2 ersetzt sind. Bei dem Vorgang der Substitution muß angesichts der Aufteilung in gebundene und freie Variablen unterschiedlich verfahren werden. Freie Variablen in dem zu substituierenden Wert können durch Substitution in einen Lambda-Term gebunden werden, wenn dieser gleichnamige formale Parameter besitzt (Namenskollision). Zum Beispiel würde die freie Variable y des Terms $\lambda(x)(y\,x)$ bei folgender Substitution eingefangen werden:

$$(\lambda(y)(f\ y))[\lambda(x)(y\,x)/f]$$

Um dies zu vermeiden, müssen alle Vorkommen des kollidierenden Bezeichners in dem Lambda-Term vor der Substitution umbenannt werden (alpha-Konversion). Die Fälle der Substitution sind die folgenden (nach [Reade 89]):

$$
\begin{aligned}
v[E/v] &= E \\
x[E/v] &= x \; ; x \neq v \\
(E_1 E_2)[E/v] &= (E_1[E/v])(E_2[E/v]) \\
(\lambda(v)E_1)[E/v] &= (\lambda(v)E_1) \\
(\lambda(x)E_1)[E/v] &= \lambda(x)(E_1[E/v]) \; ; \; x \neq v \; \wedge \; \neg freeIn(E,x) \\
&= \lambda(y)((E_1[y/x])[E/v]) \; ; \; x \neq v \; \wedge \; freeIn(E,x) \; \wedge \; \neg freeIn(E_1 E, y)
\end{aligned}
$$

Mit Hilfe der folgenden Regel (Eta-Reduktion) kann eine Abstraktion vereinfacht werden, wenn der formale Parameter lediglich an eine Applikation weitergereicht wird:

$$\lambda(v)(E\ v) \quad \xrightarrow{\eta} \quad E \; ; \; \neg freeIn(E, v)$$

Die Reduktionsregeln des Lambdakalküls gelten auch für CPS. In Verbindung mit Meta-Evaluierungsfunktionen, die primitive Operationen auf konstante Argumente anwenden, lassen sich durch Beta-Eta-Reduktion eine Reihe von Standardoptimierungen verwirklichen:

- Das Falten konstanter Ausdrücke (*constant folding*) erfolgt durch die Meta-Evaluierung von TML-Termen, z.B.:

  (+ 2 3 **cont**(t) ⋯) ⟶ (**lambda**(t) ⋯) 5

- Die Verbreitung von Konstanten und Kopien (*copy propagation*) erfordert in TML keine Seiteneffektanalyse, weil alle Bezeichner an unveränderliche Werte gebunden sind. Veränderliche Variablen werden wie in der denotationalen Semantik [Stoy 77] in zwei Komponenten aufgeteilt: eine unveränderliche Bindung an eine Speicherzelle und explizite Zugriffsoperationen auf die Speicherzelle. Die eigentliche Verbreitung von Werten erfolgt durch Beta-Reduktion; beispielsweise:

  (**lambda**(x)
  (+ x 1 **cont**(t) ⋯) ⟶ (+ 3 1 **cont**(t) ⋯)
  3)

- Die Funktionsexpansion (*inlining*) stellt für Sprachen mit sehr hohem Funktionsaufkommen vielleicht die wichtigste Optimierung dar. Die Strukurierung eines Programms in Klassen oder abstrakte Datentypen wirkt sich in einer hohen Aufrufdichte aus, da möglichst viel Funktionalität und häufig vorkommende Codemuster wie Iterationen, Selektionen, Listenoperationen usw. geerbt oder über generische Bibliotheken importiert werden. Codevereinfachungen und Optimierungen können nicht über Funktionsaufrufe

hinweg erfolgen. Erst durch Funktionsexpansion entstehen größere Sequenzen von primitiven Operationen, an denen obige Vereinfachungen greifen können. Zusätzlich entfällt der Zusatzaufwand des Funktionsaufrufs, und in manchen Fällen erübrigt sich auch die Erzeugung eines Funktionsobjekts. Auch die Funktionsexpansion läßt sich als Beta-Reduktion realisieren:

(**lambda**(f)
  (f 2 **cont**(t) $\cdots$) $\longrightarrow$ (**lambda**(x c) (c x) 2 **cont**(t) $\cdots$)
proc(x c)(c x))

Die in diesem Abschnitt geschilderten Optimierungsverfahren werden in dem TooL-Optimierer eingesetzt. Der TML-Code, der direkt aus dem TooL-Syntaxbaum gewonnen wird, eignet sich aber nicht ohne weitere Verfahren für obige Transformationen, da jede einzige Operation als Methodenaufruf codiert ist. Weitere Mechanismen, die den in Kapitel 4 beschriebenen ähneln, müssen eingesetzt werden, um die Abstraktionshürden der dynamischen Methodenaufrufe zu überwinden. Diese Verfahren werden im nächsten Kapitel beschrieben.

6. Der TooL-Optimierer

Die Darstellungen der letzten beiden Kapitel bilden eine Grundlage für die Entwurfsentscheidungen des TooL-Optimierers. Die Eckpunkte des Entwurfs lassen sich wie folgt charakterisieren:

- Ein Profiler soll Empfänger- und Argumentklassen, Selektoren und Aufrufhäufigkeiten zur Laufzeit von Programmen festhalten. Diese Information läßt sich aufgrund der persistenten Natur des TooL-Systems über mehrere Sitzungen hinweg gewinnen.
- Der Aufbau eines Methodencaches mit empfänger- und argumentspezifisch optimierten Methoden wird durch die Profilinformation gesteuert und nutzt auch die Information über Argumentklassen aus. Dieser Vorgang verläuft wegen der hohen Laufzeit des in TL geschriebenen TooL-Optimierers nicht dynamisch, sondern muß explizit ausgelöst werden. Auch der Methodencache bleibt persistent erhalten.
- Weder selektorbasierte Klassenabschätzung noch Inline-Caching werden eingesetzt, denn die dadurch erzwungene Mitbehandlung der Sonderfälle, in denen Klassentests fehlschlagen, führt zu großen Codezuwächsen und erfordert einen Codeverschiebungslauf (*splitting*). Statt dessen wird das Typsystem von TooL ähnlich wie in [Atkinson 86] und [Johnson et al. 88] als Informationsquelle angesehen und das Problem der Mitführung der Sonderfälle vermieden.
- Eine Klassenanalyse soll Klasseninformation bei einem Durchlauf durch den TML-Graphen verbreiten und dabei dynamische Methodenaufrufe durch statisch gebundene Funktionsaufrufe ersetzen.

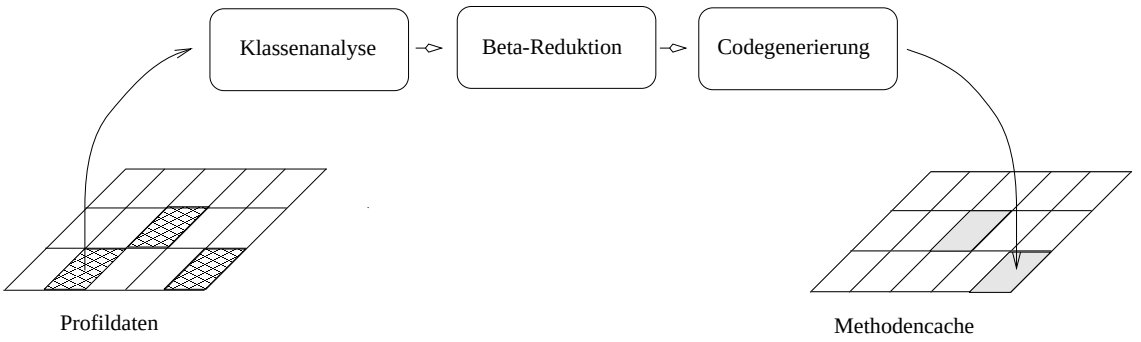


Abbildung 6.1: Die Phasen des TooL-Optimierers

- Der in [Kiradjiev 94] beschriebene CPS-Optimierer für TML dient als Inlining-Mechanismus und wird erweitert, um allgemein Endrekursionen in Sprünge umwandeln zu können. Damit sollen den in TooL auftretenden Nachteilen der reinen Objektorientierung auf möglichst saubere Weise begegnet werden, denn so brauchen auch die Kontrollstrukturen while, for usw. keine festverdrahtete Sonderbehandlung.

Die Phasen des TooL-Optimierers sind in Abbildung 6.1 angedeutet.

6.1 Profiling

Eine der Kernaussagen der Arbeit von Hölzle [Hölzle 94] lautet: Klasseninformation, die in objektorientierten Sprachen zur Übersetzungszeit nur durch mühsame und wenig effektive Analysen gewonnen werden kann, liegt während der Laufzeit zu Genüge vor. Ein wirkungsvoller Methodencache mit empfängerspezifisch übersetzten Methoden kann nur durch Laufzeitinformation über tatsächlich aufgetretene Empfängerklassen aufgebaut werden. Daher bietet sich der Einsatz eines Profilers an, der Information über das Laufzeitverhalten von Programmen sammelt. Die Phasen des Profilings und der Optimierung sind zeitlich getrennt, da der Programmierer oder die Programmiererin das Profiling und die Optimierungsläufe explizit auslösen müssen. Wie in Abschnitt 4.4 diskutiert, lassen die Messungen von [Grove et al. 95], die eine hohe Korrelation zwischen Laufzeitprofilen verschiedener Programmversionen über größere Zeiträume hinweg nachweisen, den Modus der getrennten Profiler- und Optimiererläufe in TooL als gerechtfertigt erscheinen.

In TooL werden zur Laufzeit anfallende Klasseninformationen in einer Profiltabelle festgehalten. Hierzu wird die virtuelle Maschine des Tycoon-Systems folgendermaßen erweitert: bei der Ausführung eines send-Bytecodes liegen der Selektor der Methode, der Empfänger und die sonstigen Argumente im Laufzeitkeller. Die Klassen des Empfängers und der Argumente werden mit einem Aufrufzähler in eine Hashtabelle eingetragen, deren Einträge die in Abbildung 6.2 angedeutete Struktur besitzen.

key	receiver	selector	count	classes(arg1)	classes(arg2)

Abbildung 6.2: Struktur eines Profileintrags

Gestützt von der empirischen Feststellung, daß in Smalltalk mehr als 95% aller Botschaften mit drei oder weniger Argumenten aufgerufen werden (der Empfänger wird auch als Argument angesehen) [Krasner 83], wird die Anzahl der zusätzlichen Argumente pro Eintrag in der Profiltabelle auf zwei begrenzt; somit werden die Klassen von bis zu drei Argumenten festgehalten. Für jedes Argument werden nur bis zu zwei mögliche verschiedene Klassen aufgezeichnet, denn mehr Klassen deuten einen zu hohen Grad an Polymorphismus an, so daß die gesammelten Klasseninformationen für dieses Argument keine geeigneten Hinweise an den Optimierer darstellen.

Um den Aufwand des Profilings möglichst klein zu halten, wird eine einfache Hashing-Strategie mit folgender Hashfunktion eingesetzt:

$$h(selector, class) = ((class * const) + selector) \; mod \; hashsize,$$

Die Tabelle wird durch offene Adressierung gefüllt; bei Kollisionen wird linear sondiert, bis ein freies Feld vorliegt. Die Länge der Sondierung ist begrenzt und im Falle einer gefüllten Sondierungssequenz wird der Eintrag mit der kleinsten Zahl von Aufrufen überschrieben. Da TooL-Programme von einem Bytecodeinterpreter bearbeitet werden, und ein dynamischer Methodenaufruf mehrere C-Funktionsaufrufe und Hashoperationen umfaßt, fällt der Zusatzaufwand des Profilings nicht unverhältnismäßig hoch ins Gewicht. Messungen an nicht optimierten TooL-Programmen mit einer hohen Botschaftsdichte zeigen, daß die für das Profiling benötigte Zeit nicht mehr als 10% der Gesamtlaufzeit von Programmen ausmacht.

6.2 Aufbau eines empfänger- und argumentspezifisch optimierten Methodencaches

Die Aufrufzähler in den Einträgen der Profiltabelle dienen als Hinweise, bei welchen Methoden eine Optimierung sich lohnen könnte, denn durch sie sind besonders oft aufgerufene Methoden auffindbar. Bedingt durch die Tatsache, daß die Profilinformation keine aufrufortspezifischen Häufigkeiten beinhaltet[1], wird in der jetzigen Version des TooL-Optimierers diese Information nur auf einfache Weise ausgenutzt: Methoden mit einer Aufrufhäufigkeit (normiert um die maximale Zahl der Aufrufe) größer als eine Grenze θ werden als Optimierungskandidaten angesehen; andere werden nicht optimiert. Messungen haben ergeben, daß bei dem Wert von 0.001 für θ circa die Hälfte der Profileinträge wegfällt, und daß die Effizienzsteigerung durch den Cache von optimierten Methoden 90% bis 95% der maximal möglichen Effizienzsteigerung bei $\theta = 0$ ist.

Für jeden Optimierungskandidaten läßt sich aus dem Selektor und der Empfängerklasse die Repräsentation einer Methoden auffinden. In dieser sind neben dem ausführbaren Code zusätzlich der TML-Code[2] und die Signatur der Methode abgelegt. In der Signatur stehen die Typen der Argumente und des Ergebnisses der Methode. Diese Information wird nur für einen begrenzten Satz von Basistypen wie Int, Fun, Bool, String, und Char ausgenutzt, um die Klassen der Argumente zu bestimmen. Die Information aus der Signatur, bereichert um die Profildaten über die Klassen des Empfängers und der Argumente, stellt den Ausgangspunkt eines Klassenanalyseverfahrens, das auf den TML-Code der Methode angewendet wird.

Die Typangaben von Signaturen sind dabei eine wichtige Informationsquelle, deren Ausschöpfung allerdings die Subtypisierung für die Basisklassen einschränkt, wie in dem nächsten Abschnitt erläutert wird.

6.3 Markierte Typen

Wenn in TS (*Typed Smalltalk*) [Johnson et al. 88] Variablen als Exemplare eines Typs deklariert werden, kann diese Information durch einen Optimierer ausgenutzt werden, um

[1] Aufrufortspezifische Laufzeitprofile sind nach [Grove et al. 95] die geeigneteste Informationsquelle für Optimierungen, aber gemittelte Profildaten stellen einen guten Kompromiß zwischen Granularität der Information und Einfachheit des Profilings dar.

[2] Aufgrund der einfachen Abbildung von TooL-AST nach TML ließe sich an dieser Stelle statt des relativ voluminösen TML-Graphen auch eine kompakte Bytecodierung des TooL-AST eintragen, die bei Optimierungsbedarf in TML konvertiert werden kann.

eventuell auftretende dynamische Methodenaufrufe durch statische Funktionsaufrufe zu ersetzen, denn Klassen sind in TS Typen. In TooL dagegen bieten die in den Signaturen aufgeführten Typen aufgrund der strukturellen Typisierung und dem Subtyppolymorphismus keine verläßliche Aussage über tatsächlich zur Laufzeit auftretende Klassen. Dies bedeutet, daß Typangaben in den Signaturen keine Grundlage für das statische Binden von Methoden bilden, denn Exemplare von übergebenen Subtypen können eigene Methodenimplementierungen besitzen, die nicht mit der statisch gebundenen Implementierung übereinstimmen. Demzufolge können die in den Signaturen vorhandenen Typhinweise nicht zu Optimierungszwecken herangezogen werden, und aus der Sicht des Optimierers erscheint TooL wie eine gänzlich untypisierte Sprache.

In Modula-3 [Cardelli et al. 89] ist das Konzept der „markierten Typen“ (*branded types*) vorgesehen, das für einige Typen die Subtypbeziehung zu anderen Typen verhindert, um auszuschließen, daß durch Übergabe eines Subtypen verdeckte Felder eines abstrakten Datentypen zugänglich gemacht werden. Im TooL-System werden einige Basistypen wie Int, Fun, Bool, String, und Char als markierte Typen angesehen, d.h. sie können mit ihren jeweiligen Klassen gleichgesetzt werden. Von diesen markierten Typen können keine Subtypen gebildet werden, und sie sind durch ihre Bezeichner identifizierbar, so daß diese Bezeichner in den Signaturen der Methoden als Hinweise an den Optimierer dienen können. Subklassen dieser Basisklassen lassen sich weiterhin definieren.

Die Einschränkung der Subtypisierung stellt keinen großen Verlust an Modellierungsmächtigkeit dar, da es fraglich ist, ob sinnvolle Subtypen dieser Basistypen existieren. Für die Klassen Int und Char würde zum Beispiel die naheliegende, objektorientierte Programmiertechnik der Hinzunahme von neuen Methoden in Subklassen ohnehin Subklassen erzeugen, die keine Subtypen sind, denn Int und Char besitzen binäre Methoden.

Ferner ist der Gedanke einer festgelegten Semantik für Basistypen durchaus begründbar: Für die rein objektorientierte Sprache Emerald [Hutchinson 87] argumentiert Hutchinson, daß eine eindeutige Implementierung für Basistypen zwingend notwendig ist, denn das restliche System wird mit einer impliziten Annahme über die Semantik der Basistypen entwickelt, der die Korrektheit des Systems zugrundeliegt. Mehrdeutigkeiten sollten bezüglich der Semantik von Grundkonstrukten der Programmierung ausgeschlossen sein.

Die Vorteile des Ansatzes der markierten Typen sind zweifach:

- Keine Klassenabschätzungen oder speziellen Tricks sind notwendig, um die Folgen der reinen Objektorientierung zu eliminieren; auch die „harten Nüsse“ der Kontrollstrukturen lassen sich damit ohne Sonderbehandlung knacken. Ferner entfallen die Klassentests der Klassenabschätzung und die Angabe der Fälle für fehlgeschlagene Klassentests. Die Duplikation von Code für größere monomorphe Pfade (*splitting*) wird unnötig.

- Da die Information über Basisklassen in den Signaturen neben den Profildaten lediglich eine weitere Informationsquelle darstellt, passen die markierten Typen gut in das Gefüge der Klassenanalyse, in der auch noch andere Möglichkeiten zur Verbreitung von Klasseninformation ausgenutzt werden.

6.4 Klassenanalyse

Die Aufgabe der Klassenanalyse besteht darin, Klasseninformation durch einen TML-Graphen zu verbreiten und eventuell neue Klasseninformation durch einfache Inferenzen zu gewinnen. Da dabei dynamisch gebundene Referenzen an Methoden durch statisch gebundene Funktionsaufrufe ersetzt und neue Kontrollflußknoten eingeführt werden, ist das Verfahren strenggenommen ein Analyse- und Transformationsschritt. Eine analysierte Methode wird anschließend mit Hilfe der Beta-Reduktion weitergehend optimiert, wobei statisch gebundene Referenzen expandiert werden können (*inlining*).

6.4.1 Der Analyse-Algorithmus

Der Analyse-Algorithmus soll jeweils eine Zielmethode, die sich aus einem Profileintrag auffinden läßt, bearbeiten und Methodenaufrufe durch Funktionsaufrufe ersetzen. Dabei werden Hilfsfunktionen aufgedeckt, in denen ebenfalls Methodenaufrufe ersetzt werden. Das Ergebnis ist eine modifizierte Zielmethode mit allen transitiv referenzierten Hilfsfunktionen.

In dem Analyseverfahren wird Information über „bekannte Empfänger" durch den TML-Graphen einer Methode gereicht und aktualisiert. Ein *bekannter Empfänger* ist ein Tupel $<v,T>$, wobei v eine Variable und T eine Menge von möglichen Klassen ist. Zu Beginn der Analyse einer Methode ist die aufgezeichnete Information über die Parameter einer Methode (self inklusive) aus den Profildaten und der Signatur bekannt. Diese Information stellt die Menge der *initialen* bekannten Empfänger einer Methode dar, die um abgeleitete bekannte Empfänger während einer TML-Traversierung erweitert wird. Der Klassenanalyse-Algorithmus läßt sich wie folgt beschreiben:

1. Zwei Mengen von Methoden werden benötigt: eine *Agendamenge* der noch zu bearbeitenden Methoden und eine *Ergebnismenge* der analysierten Methoden. Anfangs befindet sich nur die aufgefundene Zielmethode eines Profileintrags in der Agendamenge; die Ergebnismenge ist leer. Der aktuelle Optimierungskandidat wird der Agendamenge entnommen.

2. Der TML-Code des aktuellen Optimierungskandidaten wird unter Mitführung der bekannten Empfänger traversiert. Botschaften an eindeutig bekannte Empfänger werden durch Funktionsaufrufe ersetzt, und eventuell können neue bekannte Empfänger aufgedeckt werden. Wenn eine Variable mehrere mögliche Empfängerklassen besitzt, wird ein Verzweigungspunkt in den TML-Graphen an der Stelle der Botschaft an die Variable eingefügt, so daß monomorphe Zweige mit duplizierten Methodenaufrufen entstehen. An Vereinigungspunkten im Graphen werden die Mengen der möglichen Klassen für die bekannten Empfänger vereinigt.

3. Der TML-Code einer ersetzten Botschaft wird mit seinen initialen bekannten Empfängern in die Agendamenge eingefügt. Diese Menge an initial bekannten Empfängern beinhaltet self und möglicherweise weitere Parameter, die sich aus der Signatur der ersetzten Botschaft ergeben können.

4. Eine traversierte Methode kommt in die Ergebnismenge der bearbeiteten Methoden. Der nächste Optimierungskandidat wird der Agendamenge entnommen, und obiges Verfahren wiederholt sich ab Punkt 2, bis die Agendamenge leer ist.

6.4.2 Ein Beispiel

Das Traversierungsverfahren läßt sich am Beispiel der Methode while veranschaulichen. Der TML-Graph der Methode ist wie folgt:

```
proc(self cond statement c)
  (send "[]" cond cont(t_1)
  (send "?" t_1
    proc(c_3)
     (send "[]" statement cont(t_4)
     (send "while" self cond statement cont(t_5)
     (c_3 t_5)))
    cont(t_6)
      (c t_6)))
```

Beim Eintritt in den TML-Graphen sind nur die Parameter der Methode (die initialen bekannten Empfänger) bekannt:

```
  proc(self cond statement c)
▷ (send "[]" cond cont(t_1) ... {<self,Nil> <cond,Fun:Bool>  <statement,Fun>}
```

Die Klasse der Variable cond ist der Signatur der Methode while entnommen und führt dazu, daß der Methodenaufruf ersetzt wird. Im Fall der Klasse Fun wird diese Ersetzung implizit Beta-reduziert, weil der markierte Typ Fun aufgrund der Information über den Wertebereich einer Funktion eine Sonderstellung in der tatsächlichen Implementierung besitzt. Die Beta-Reduktion wäre an diesem TML-Knoten ohnehin beim Lauf des CPS-Optimierers erfolgt. Daher werden Methodenaufrufe (send "[]" function arguments ...) an die Klasse Fun direkt in Funktionsaufrufe der Form (function arguments ...) überführt. Bei Ersetzungen von Botschaften durch Funktionen kann der Wertebereich der Funktion, der in der Signatur aufgeführt ist, Aufschluß über die möglichen Klassen der Fortsetzungsvariable der Funktion geben, sofern der Wertebereich ein markierter Typ ist. Der Wertebereich der Funktion cond ist die abstrakte Klasse Bool; in dem Klassenanalyse-Algorithmus werden die konkreten Subklassen True und False als Menge der möglichen Klassen der Variable t_1 eingetragen:

```
  proc(self cond statement c)
   (cond cont(t_1)
▷ (send "?" t_1 proc(c_3) ...    { ... <t_1,(True,False)> ... }
```

Die Variable t_1 besitzt an der Stelle des Aufrufs der Botschaft „?“ zwei mögliche Empfängerklassen, was zu einer Einfügung eines Verzweigungspunktes in monomorphe Pfade für t_1 führt:

```
proc(self cond statement c)
 (cond cont(t_1)
 (lambda(join)
  . (lambda(argument)
  .   | (classTest t_1 "True" "False"
  .   |     cont()
  .   |   ▷  (send "?" t_1 argument join)  { ... <t_1,True>  ...}
  .   |     cont()
  .   |       (send "?" t_1 argument join)) { ... <t_1,False>  ...}
  .   | proc(c_3) ...
  . cont() ...
```

Da jetzt für jeden der Pfade die Klasse von t_1 eindeutig bekannt ist, lassen sich die Methodenaufrufe an „?“ durch Funktionsaufrufe ersetzen, und die Implementierungen der Methode „?“ werden in die Agendamenge aufgenommen. Ab dem Vereinigungspunkt join wird die Analyse mit dem bekannten Empfänger <t_1,(True,False)> fortgesetzt, für den die Klasseninformation der beiden konvergierenden Pfade vereinigt wurde:

```
proc(self cond statement c)
 (cond cont(t_1)
 (lambda(join)
  . (lambda(argument)
  .   | (classtest t_1 "True" "False"
  .   |     cont()
  .   |       (True:? t_1 argument join)
  .   |     cont()
  .   |       (False:? t_1 argument join))
  .   | proc(c_3) ...
  . ▷ cont() ...    { ... <t_1,(True,False)>  ...}
```

6.4.3 Verbreitung von Klasseninformation

In dem vorherigen Beispiel der Methode while wurde an der Ersetzung der „[ ]“-Botschaft exemplarisch demonstriert, wie der Wertebereich einer ersetzten Methode zu neuer Klasseninformation führen kann (in dem Beispiel für die Variable t_1). In der Implementierung des TooL-Optimierers werden folgende Möglichkeiten zur Gewinnung und Verbreitung von Klasseninformation ausgeschöpft:

- Wertebereiche von ersetzten Methoden oder primitiven Operationen können die Klassen der durch die Fortsetzung definierten Variablen festlegen. Der folgende Methodenaufruf könnte z.B. durch einen Funktionaufruf an eine Methode mit der Signatur foo(:Self):Int ersetzt werden:

  ```
  (send "foo" self cont(t) ...
  ```

  Hieraus läßt sich inferieren, daß t den markierten Typ Int besitzt:

(Object:foo self **cont**(t) ... {... <t,Int> ...}

- Durch Lambda-Applikationen definierte Variablen können die Klassen ihrer Argumente übernehmen, anhand der Applikation

  (**lambda**(x) (...) y) { ... <y,OrderedCollection> ...}

  wird gefolgert, daß x die Klasse OrderedCollection besitzt.

- Die Variablen gebundener Fortsetzungen erhalten ihre Klassen als Vereinigung der Klassen der Argumente von Aufrufen der Fortsetzung. Für die Variable t in dem folgenden Beispiel läßt sich der bekannte Empfänger <t,(Int,OrderedCollection)> bilden:

  (**lambda**(join)
  ... (join x) {... <x,Int> ...}
  ... (join y) {... <y,OrderedCollection> ...}
  cont(t) {... <t,(Int,OrderedCollection)> ...}
  ...

- Vor der Aufnahme einer aufgedeckten Methodenimplementierung in die Agendamenge kann neben der Signatur der Methode der Aufrufkontext Hinweise über die Klassen der Formalparameter liefern. Der folgende Methodenaufruf an self deckt nicht nur eine weitere Methodenimplementierung auf, sondern liefert Information über die Klassen der Parameter:

  (send "averageBy" self 100 **cont**(t_1) ... {... <self,Statistics> ...}

  So kann zum Beispiel die Information aus der Signatur von averageBy(factor :Number):Void um den Hinweis erweitert werden, daß für diesen Aufrufkontext immer ein Literal der Sorte Int (100) für den Parameter factor vorliegt:

  proc(self factor) ... {<self,Statistics><factor,Int>}

6.4.4 Optimierung durch Beta-Reduktion

Das Ergebnis der Klassenanalyse ist eine Menge von allen transitiv über self-Botschaften referenzierten Methoden und sonstigen bekannten Referenzen, etwa aus Botschaften an Parameter der Methode. In den Rümpfen dieser Methoden sind die dynamischen Referenzen an eindeutig bekannte Empfänger durch statische Funktionsaufrufe ersetzt worden. Da implizite, möglicherweise wechselseitige Rekursionen dadurch in explizite rekursive Referenzen umgewandelt worden sind, werden alle aufgedeckten Methoden in einem Y-Primitiv zusammengefaßt. Das Y-Konstrukt der Methode while, in dessen Rumpf die Methoden True:? und False:? ersetzt wurden, ist:

```
proc(c_0)
 (Y proc(c_1 False:? True:? Nil:while)
 (c_1
  cont()
   (c_0 Nil:while)
  proc(self ifTrue c_2)          ; Rumpf von False:?
   (c_2 ok)
  proc(self ifTrue c_3)          ; Rumpf von True:?
   (ifTrue cont(t)
   (c_3 t))
  proc(self cond statement c_4)  ; Rumpf von Nil:while
   (cond cont(t)
     ...
       ... )))))))
```

Die Beta-Reduktion erfolgt auf diesem Y-Primitiv und expandiert kurze Methoden an ihren Aufruforten. Das Ergebnis dieser Optimierungsphase ist eine Methode, die auf bestimmte Empfänger- und Argumentklassen spezialisiert ist. Die spezialisierte Methode wird anschließend übersetzt und ergibt damit einen Funktionsabschluß mit allen referenzierten Hilfsfunktionen, der in den Methodencache eingetragen wird. Ein ungelöstes Problem ist hierbei, daß bei der Optimierung mehrerer Methoden die gleichen Hilfsfunktionen wiederholt übersetzt und angelegt werden. Eleganter wäre der Einsatz eines Caches von Hilfsfunktionen.

6.4.5 Aufruf einer spezialisierten Methode

Eine spezialisierte Methode kann möglicherweise besondere Argumentklassen erwarten, denn der Optimierer kann Botschaften an Argumente durch statische Funktionsaufrufe ersetzt haben. In einem solchen Fall ist das Ergebnis der Optimierungsphase nicht nur auf die Empfängerklasse, sondern auch auf eine oder mehrere der Argumentklassen zugeschnitten.

Wenn Ersetzungen aufgrund von Profildaten und nicht von Signaturangaben erfolgten, können durchaus unerwartete Argumentklassen auftreten. Da der Methodenaufruf ausschließlich von der Empfängerklasse und dem Selektor bestimmt wird, muß zur Laufzeit geprüft werden, ob die Argumentklassen mit denen übereinstimmen, für die eine Methode spezialisiert wurde. Sind alle Argumente wie erwartet, kann die optimierte Version der Methode aufgerufen werden. Im anderen Fall muß die ursprüngliche (nicht-spezialisierte) Version ablaufen. An dem folgenden Beispiel ist dieser Mechanismus erkennbar:

```
proc(self aStream c_1)
  (lambda(specializedLabel)
    (classTest aStream "File"
      cont()
        (specializedLabel)
      cont()
        (<oid 0x2C79FC> self aStream c_1)
    cont()
      ...
      (File:put aStream ch cont(t)
        ...)))
```

Der Code entstammt der Methode printOn, die auf die Empfängerklasse Set und die Klasse File für das erste Argument aStream zugeschnitten ist. Der Aufruf der ursprünglichen Methode erfolgt über eine explizite Objektreferenz (<oid 0x2C79FC>) an den Bytecode.

6.5 Elimination von Endrekursion

Der im vorherigen Abschnitt beschriebene Algorithmus der Klassenanalyse und der Beta-Reduktionsmechanismus erweisen sich vor allem durch die Einführung der markierten Typen als wirkungsvolle Optimierungsstrategie. Eine wesentliche Optimierungstechnik, um die der in [Kiradjiev 94] beschriebene TML-Optimierer erweitert wurde, ist die Überführung von Endrekursionen in Iterationen. Diese ist nötig, um die Leistung von explizit codierten Kontrollstrukturen zu erreichen, denn endrekursive Schleifen besitzen zwei Nachteile:

1. Funktionsaufrufe sind erheblich langsamer als einfache Sprünge. Dies gilt sowohl für virtuelle Maschinen wie die TVM, in der ein `call`-Opcode interpretiert wird, als auch für reale Prozessoren. Gerade Schleifen zählen zu den häufig ausgeführten Teilen eines Programms und sollten nicht durch teure Kontrollflußmechanismen zusätzlich verlangsamt werden.

2. Die Speicheranforderungen einer rekursiven while- oder for-Schleife mit einer großen Iterationstiefe sind unnötig hoch, da alle Aktivierungssegmente bis zum Rekursionsende im Keller gespeichert werden müssen. Endlose Ereignisschleifen sind mit endrekursiven Kontrollstrukturen unmöglich[3].

Aus diesen Gründen sehen andere rein objektorientierte Sprachen fest verdrahtete Schleifen oder eingebaute Schleifenprimitive vor. In SELF wird neben dem dynamischen Methodenaufruf und der Vererbung ein Schleifenprimitiv als Grundbaustein der reinen Objektorientierung angesehen [Chambers 92]. Da TooL keine derartige Konstrukte vorsieht, ist es umso entscheidender, daß Endrekursion eliminiert wird.

Der im folgenden dargestellte Ansatz unterscheidet sich von dem in [Steele 78] beschriebenen Verfahren. Steele analysiert Funktionen daraufhin, ob sie ihre Definitionsumgebung überleben können (etwa als Rückgabewert einer anderen Funktion). Sollte dies nicht der Fall sein,

[3] TL besitzt Schleifenkonstrukte, die explizit im AST in Sprünge übersetzt werden. Daher stellt der Einsatz von TL-Programmen als Threads oder Ereignisschleifen kein Problem dar.

werden alle Aufrufe an diese Funktion einfach in Sprünge übersetzt, sofern keine der Aufrufe in anderen Aufrufen geschachtelt ist. Wechselseitige Rekursionen brauchen auf diese Weise keine besondere Behandlung und können ohne weiteres auf Iterationen abgebildet werden. Die in dieser Arbeit verfolgte Methode kommt ohne obige Analyse aus, und auch endrekursive Funktionen, die ihre Definitionsumgebung überleben, können optimiert werden. Allerdings entfällt die Möglichkeit, wechselseitige Rekursionen auf einfache Art zu optimieren.

Die wesentlichen Aspekte des Ansatzes dieser Arbeit zur Überführung von Endrekursionen in Iterationen sind:

- Erkennung von Endrekursion: Endrekursionen sind nach einem Reduktionslauf daran erkennbar, daß alle Fortsetzungen der rekursiven Aufrufe direkt die Rücksprungfortsetzung anspringen.
- Erweiterung von TML um Fortsetzungen mit mehr als einem Argument: Die Zwischensprache TML wird derart erweitert, daß Fortsetzungen zwei oder mehr Argumente übernehmen können, wenn der Aufruf der Fortsetzung explizit erfolgt. Für implizit aufgerufene anonyme Fortsetzungen (Lambda-Fortsetzungen) ergeben mehr Argumente keinen Sinn, da diese Fortsetzungen nur ein Ergebnis der letzten Berechnung binden. In der virtuellen Maschine wird beim Sprung an eine Fortsetzung der aktuelle Kellerrahmen von überflüssigen Zwischenergebnissen durch eine Anpassungsoperation bereinigt. Im Fall eines Sprungs an eine Fortsetzung mit mehreren Argumenten müssen in der TVM einige der oberen Keller-Einträge über diese Anpassungsoperation hinaus als Argumente an die Fortsetzung erhalten bleiben. Dieses ist in Abbildung 6.3 dargestellt.
- Entfernen von unveränderlichen Argumenten: Wenn einige Parameter einer Endrekursion unverändert an alle endrekursiven Aufrufe übergeben werden, können diese Parameter aus dem Kopf und den Aufrufen der endrekursiven Funktion entfernt werden. Dadurch werden Schleifen vereinfacht und beschleunigt, weil die redundante Auswertung der Argumente nicht mehr stattfindet. Zusätzlich begünstigt dieses die Integration von Funktionsparametern, da weniger Referenzen auf den Parameter bestehen. Auf diese Weise kann zum Beispiel eine while-Schleife mit ihren Block-Argumenten vollständig vereinigt werden, und das Anlegen der Blöcke für die Bedingung cond und den Schleifenrumpf statement entfällt.

Gemäß dieser Überlegungen wird der TML-Optimierer um einen Mechanismus erweitert, der durch folgende Transformation endrekursive Funktionen in nichtrekursive Funktionen überführt, in der die ursprüngliche Funktion als rekursive Fortsetzung auftritt:

```
                                  proc(V1 ... Vn ce cc)
                                    (Y proc(c recurse)
                                    (c
proc(v1 ... vn ce cc)                 cont()
  ( ... )                  ⟶           (recurse V1 ... Vn)
                                      cont(v1 ... vn)
                                        ( ... ) [ recurse/ tailrecursive call]
```

Nach der Transformation sind in dem Rumpf der ursprünglichen Funktion alle rekursiven Funktionsaufrufe durch Aufrufe an die eingeführte Fortsetzung recurse substituiert. Aufrufe

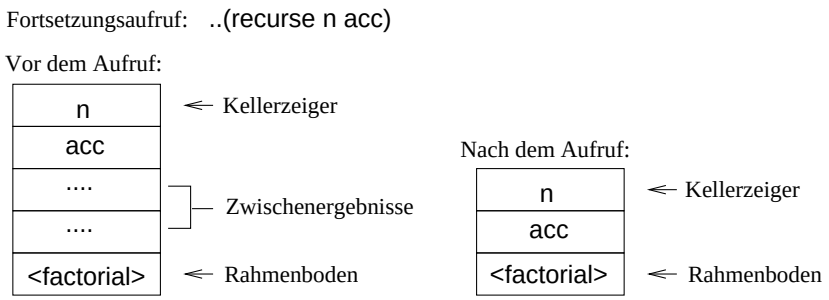


Abbildung 6.3: Kellerrahmen vor und nach einem Sprung an eine Fortsetzung mit mehreren Argumenten

an Fortsetzungen werden bei der Codegenerierung (siehe Kapitel 7) in Sprünge übersetzt. Als Beispiel für diese Transformation soll die Optimierung einer endrekursiven Fakultätfunktion dienen:

```
proc(n acc e c_1)
  (== n 1
    cont()
     (c_1 acc)
    cont()
     (- n 1 e cont(t_1)
     (* n acc e cont(t_2)
     (fact t_1 t_2 e c_1))))
```

⟶

```
proc(N ACC e c_1)
  (Y proc(c_2 recurse)                    ; (1)
  (c_2
    cont()
     (recurse N ACC)                      ; (2)
    cont(n acc)                           ; (3)
     (== n 1
       cont()
        (c_1 acc)
       cont()
        (- n 1 e cont(t_1)
        (* n acc e cont(t_2)
        (recurse t_1 t_2))))))            ; (4)
```

An diesem TML-Code sind folgende Punkte erkennbar:

1. Aus einer ehemals rekursiven Funktion wurde eine nicht-rekursive Funktion. Das Y-Primitiv im Rumpf definiert lediglich eine rekursive Fortsetzung recurse, die in eine Sprungmarke übersetzt wird.
2. An den ersten Aufruf der Fakultätfortsetzung werden zwei Argumente übergeben: die eingeführten Variablen N und ACC.
3. Die rekursive Fortsetzung besitzt zwei Parameter: n und acc.
4. In der Kellermaschine TVM erfordert der Aufruf der Fortsetzung die in Abbildung 6.3 angedeutete Anpassung des Kellerrahmens, um Zwischenergebnisse vom Keller zu entfernen und die aktuellen Parameter t_1 und t_2 zu erhalten.

Die nächste Transformation eliminiert die Parameter aus der Fortsetzung, die an jeder Aufrufstelle unverändert weitergegeben werden:

```
proc(V1 ... Vn ce cc)                    proc(V1 ... Vn ce cc)
 (Y proc(c recurse)                       (Y proc(c recurse)
 (c                                       (c
    cont()                                   cont()
     (recurse V1 ... Vn)          ⟶           (recurse V1 ... Vi-1 Vi+1 ... Vn)
    cont(v1 ... vn)                          cont(v1 ... vi-1 vi+1 ... vn)
     ( ...                                    ( ...
        (recurse ... vi ... ))                   (recurse ... /i ... )) [Vi/vi]
```

Die Anwendung der beiden Transformationen auf die while-Schleife liefert folgendes Ergebnis:

```
                                         proc(self COND STATEMENT c_1)
                                         (Y proc(c_2 recurse)
                                          (c_2
                                           recurse                          ; (1)
proc(self cond statement c_1)             cont()
  (cond cont(t_1)                           (COND  cont(t_1)                ; (2)
  (classTest t_1 "True" "False"    ⟶       (== t_1 true false              ; (3)
     cont()                                  cont()
       (statement cont(t_2)                    (STATEMENT cont(t_2)
       (Int:while self cond statement c_1))    (recurse))                    ; (4)
     cont()                                  cont()
       (c_1 ok))))                             (c_1 ok)))))
```

Zu den Punkten einige Erläuterungen:

1. Die Fortsetzung des Y-Primitivs dient als Einstieg in die Schleife, die an dieser Stelle angesprungen wird.
2. Die Parameter an die ürsprungliche rekursive Funktion konnten aus der rekursiven Fortsetzung herausgezogen werden, da diese an jedem Aufruf unverändert weitergegeben werden. In dem Rumpf stehen statt der ursprünglichen Parameter die substituierten, neu eingeführten Parameter COND und STATEMENT.
3. Das classTest-Primitiv mit den Klassen True und False wurde mit Hilfe einer Meta-Evaluationsfunktion (Faltfunktion) durch das TML-Konditional ersetzt.
4. Der Sprung erfolgt in diesem Beispiel ohne Parameter.

Der TML-Graph dieser transformierten while-Schleife entspricht dem handcodierten TML-Baum, der in dem TL-Übersetzer für das Schlüsselwort while erzeugt wird. Die Optimierungsmechanismen dieser Arbeit haben eine rein objektorientiert definierte Methode in Code überführt, der äquivalent zu einer herkömmlichen Kontrollstruktur einer imperativen Sprache ist. Endrekursiv formulierte Iteratoren wie some, every und auch for lassen sich auf die gleiche Weise optimieren. Weitere Reduktions- und Expansionsläufe können Kontrollstrukturen vollständig an ihren Aufruforten integrieren. Als Beispiel soll folgendes Vorkommnis einer while-Schleife in der Methode printOn der Klasse Int betrachtet werden:

```
...
  while({divisor <= value}, {divisor := divisor * 10}),
...
```

Die Klassenanalyse transformiert den TML-Code dieses TooL-Fragments in einen TML-Baum, in dem die Botschaft while an den Empfänger self durch einen Funktionsaufruf Int:while ersetzt ist:

```
...
  (Int:while self
    proc(c_1)
      ([] divisor 0 cont(t_2)
      ([] value 0 cont(t_3)
      (send "<=" t_2 t_3 cont(t_4)
      (c_1 t_4))))
    proc(c_5)
      ([] divisor 0 cont(t_6)
      (send "*" t_6 10 cont(t_7)
      ([]:= divisor 0 t_7 cont()
      (c_2 t_7))))
    cont(t_8)
     ...
```

Durch Beta-Reduktion kann die while-Methode optimiert und an der Aufrufstelle integriert werden. Das Ergebnis ist wie folgt:

```
...
  (Y proc(c_10 recurse)
  (c_10
   recurse
   cont()
     ([] divisor 0 cont(t_2)
     ([] value 0 cont(t_3)
     (send "<=" t_2 t_3 cont(t_4)
     (== t_4 true false
       cont()
         ([] divisor 0 cont(t_6)
         (send "*" t_6 10 cont(t_7)
         ([]:= divisor 0 t_7 recurse)))
       cont()
         (c_11 ok)))))))
      ...
```

Die in diesem Kapitel vorgestellten Optimierungstechniken eliminieren die Nachteile der reinen Objektorientierung, insbesondere für Kontrollstrukturen. Die Umformung von Endrekursionen in Sprünge stellt eine Notwendigkeit für jeden ernsthaften Einsatz von TooL-Programmen dar, und erzeugt auf allgemeine Weise ohne hartverdrahtete Information (etwa über die Selektoren for und while) aus relativ ineffizienten Methoden schnelle und platzsparende iterative Schleifen. Ohne die Verwendung der markierten Typen wäre die Optimierung der Kontrollstrukturen wesentlich komplizierter unter Verwendung von Klassenabschätzung ausgefallen.

Mit Hilfe der Profilinformation kann ein Methodencache aufgebaut werden, in dem speziell auf die Empfängerklasse optimierte Methodenversionen eingetragen werden. In einzelnen Fällen sind diese Versionen zusätzlich noch auf die häufigsten Klassen der ersten beiden Argumente eingestellt, und somit wurde die Technik der empfängerspezifischen Übersetzung auf eine empfänger- und argumentspezifische Übersetzung erweitert.

Die rein objektorientierte Natur der Sprache bleibt erhalten. Auch von den Basisklassen Int, Char, Bool, Fun und String können Subklassen abgeleitet werden. Der Methodencache wird zusammen mit der Profiltabelle gelöscht, wenn Klassen redefiniert oder neue Klassen in den Klassengraphen eingefügt werden. Damit spiegeln Änderungen im Klassensystem sich ohne Einschränkungen im Laufzeitverhalten wider.

7. Codegenerierung für eine virtuelle Registermaschine

Eine Voraussetzung für die innovativen Möglichkeiten des Tycoon-Systems wie die Programmierung mit persistenten Threads [Matthes, Schmidt 94] oder mobilen Agenten in heterogenen Netzen [Mathiske et al. 95] ist eine maschinenunabhängige und effektive Ausführungsrepräsentation. Der Einsatz einer virtuellen Maschine ist eine Möglichkeit, Portabilität zu gewährleisten; dieser Ansatz wird für eine Vielzahl von Sprachen mit Erfolg genutzt, z.B. Smalltalk [Goldberg, Robson 83], Java [Gosling, McGilton 95], PC-Scheme [Bartley, Jensen 86] , TL [Matthes, Schmidt 92] und TooL [Gawecki, Matthes 96].

Die in diesen Systemen eingesetzten virtuellen Maschinen interpretieren einen vereinfachten Maschinencode, der eine gezielte Unterstützung der Semantik der Hochsprache anbietet und Hardwaredetails verbirgt. Durch Portierung des Interpreters und des Laufzeitsystems kann eine neue Hardwareplattform auf relativ einfache Weise erschlossen werden. Ferner besteht die Möglichkeit, Schnittstellen für externe Systeme in der Architektur der virtuellen Maschine vorzusehen und dadurch die flexible Integration bestehender und zukünftiger Dienste zu ermöglichen.

Einen gravierenden Nachteil stellt jedoch die Performanz von abstrakten Maschinen dar: die Ausführungsgeschwindigkeit von virtuellem Maschinencode ist um eine bis zwei Größenordnungen schlechter als die von echtem Maschinencode. Die Ausführungszeiten sind im wesentlichen durch folgende Faktoren bestimmt:

- Der Overhead einer Interpretationsschleife liegt zwischen 10 bis 50 Zyklen pro Befehl der virtuellen Maschine[1].
- Reale Maschinenregister können nicht für Operanden verwendet werden.

Der interpretierte Kellermaschinencode der TVM bietet wenig Möglichkeiten für Verbesserungen durch Codegenerierungsmaßnahmen. Statt eines neuen Entwurfs einer veränderten Kellermaschine wird im Rahmen dieser Arbeit die bestehende Kellermaschine TVM des Tycoon-Systems durch eine Registermaschine ersetzt, um einen Vergleich zwischen den beiden Rechnerarchitekturen vor dem Hintergrund des Einsatzes als virtuelle Maschine zu bieten. Die Motivation für diesen Schritt ist folgendermaßen: Registermaschinencode kann durch Registerallokation in der Codegenerierungsphase optimiert werden, weil über eine größere Menge an frei zugängigen lebendigen Operanden (*working set*) in den simulierten Registern

[1] Diese Zahlen ergeben sich aus dem SPARC-Assembler-Text des Interpreters für die Tycoon-Registermaschine, und sind nur als grobe Schätzung anzusehen.

verfügt wird, und damit explizite Operandenbewegungen vermieden werden. Techniken der Codegenerierung für Registermaschinen und der Registerallokation sind erprobt und können direkt umgesetzt werden.

Der nächste Schritt zur Effizienzsteigerung ist die Erzeugung von echtem Maschinencode bedarfsweise zur Laufzeit; Ansätze hierzu sind unter anderem in [Lee, Leone 96] und [Engler, Proebsting 94] beschrieben. Für das Tycoon System werden folgende Alternativen in Betracht gezogen:

1. Erzeugung von C-Code aus dem Bytecode und anschließende Übersetzung zur Laufzeit.

2. Direkte Erzeugung von Maschinencode aus dem Bytecode.

Für ein derartiges Vorhaben kann eine virtuelle Registerarchitektur eher geeignet sein als eine Kellermaschine, da explizit angegebene Operanden in echte Maschinenregister gelegt werden können. Da aber eine Maschinencodegenerierung zunächst experimentellen Charakter besäße, ist nach wie vor die Leistung der interpretierenden Maschine von Vorrang, wie auch die Robustheit und Einfachheit der Codegenerierung. In den folgenden Abschnitten werden die Mechanismen der Erzeugung von Registermaschinencode beschrieben. Ein Leistungsvergleich zwischen interpretierenden Keller- und Registerarchitekturen findet sich im Kapitel 8.

7.1 Die Architektur der Tycoon Registermaschine

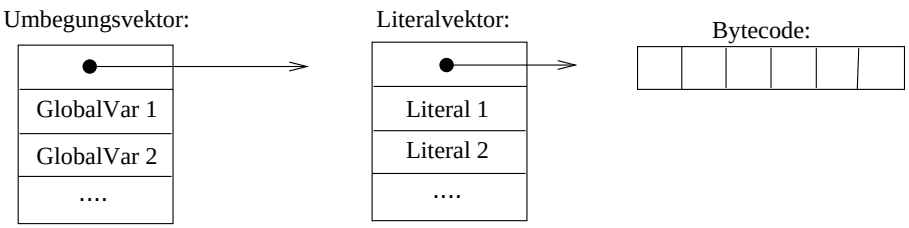


Abbildung 7.1: Format eines Funktionsabschlusses

Ausführungsgrundlage der Registerarchitektur, deren Entwurf von Andreas Gawecki stammt, ist das sogenannte *flat closure*-Format (siehe Abbildung 7.1) für Funktionsabschlüsse, auf das einige der Instruktionen der Registermaschine zugeschnitten sind. Ein Funktionsabschluß besteht in dem Tycoon-System aus zwei Vektoren: dem Vektor der freien (globalen) Bindungen einer Funktion und einem Codeobjekt bestehend aus Bytecode und einem Vektor der Literale (Zeichenketten, große Zahlen usw.). Dieser Aufbau stellt eine Möglichkeit dar, die lexikalischen Sichtbarkeitsregeln der Hochsprache zu unterstützen. Der Vorteil dieser Struktur gegenüber anderen Mechanismen wie verkettete Umgebungen (z.B. beschrieben in [Connor et al. 90]) liegt in dem schnellen Zugriff auf die freien Bindungen einer Funktion. Bei übersetzten TooL-Methoden, die noch nicht optimiert wurden, entfallen die globalen Bindungen, da Werte von Exemplarvariablen über Botschaften referenziert werden. Ein Funktionsabschluß ist die Ausführungsrepräsentation für eine Abstraktion.

Opcode (Hex)	*Befehl*	*Bedeutung*
0i	`push`	Register i im Keller ablegen
1i	`drop`	i Kellerfelder entfernen
2i	`jump`	Endrekursiver Sprung an das Codeobjekt in r0
3i	`call`	Aufruf des Codeobjekts in r0
4i	`return`	Return, i Kellerfelder entfernen
5d ii	`ld_sp`	rd $\leftarrow$ Kellerfeld ii
6d ii	`ld_lit`	rd $\leftarrow$ Literal ii
7i ds	`seti`	rd[i] $\leftarrow$ rs
80 ds	`get`	rd $\leftarrow$ rd[rs]
8x ds	`alu`	rd $\leftarrow$ rd $\otimes$ rs,
		$\otimes \in \{+, -, \times, \div, mod, \wedge, \vee, xor, shl, shr, =, >, \geq, <, \leq\}$
90 ds	`geti`	rd $\leftarrow$ rd[i]
9x di	`alui`	rd $\leftarrow$ rd $\otimes$ i,
		$\otimes \in \{+, -, \times, \div, mod, \wedge, \vee, xor, shl, shr, =, >, \geq, <, \leq\}$
ad ii	`closure`	rd $\leftarrow$ neues Feld der Größe ii
bi jj	`send`	send r0 Selektor: jj, Argumente in r1 ...r15
ci ii jj	`sendLong`	send r0 Selektor: iijj,Argumente in r1 ...r15
...		
d1 d-	`ld_ok`	rd $\leftarrow$ ok (TSP-Wert für nil)
d2 d-	`ld_false`	rd $\leftarrow$ false
d3 d-	`ld_true`	rd $\leftarrow$ true
e0 di	`ld_int`	rd $\leftarrow$ i
e1 ds	`mov`	rd $\leftarrow$ rs
...		
f1 ii	`branch`	Sprung um ii Bytes
f2 ii	`branch_cf`	Bedingter Sprung um ii Bytes

Abbildung 7.2: Ausschnitt aus dem Instruktionssatz der Registermaschine

Ein Teil des Instruktionssatzes der Registermaschine TRM[2] ist in Abbildung 7.2 wiedergegeben. Die Maschine ist eine Zwei-Adreß-Maschine mit 16 Registern, einem Keller und einem kompakten Befehlssatz (Bytecode) und bietet eine ähnliche hochsprachliche Unterstützung wie die TVM (`closure`, `send`, `class_new` usw.). Funktionsparameter werden in den Registern R1 bis R15 übergeben, wobei der aufgerufene Funktionsabschluß bzw. der Empfänger einer TooL-Botschaft in R0 abgelegt wird. Mehr Parameter dürfen nicht übergeben werden. Da der aufgerufene Funktionsabschluß in R0 übergeben wird, bleiben die globalen Bindungen während der Ausführung einer Funktion erreichbar. Per Konvention liegen Funktionsergebnisse nach einem Aufruf in R0 vor.

Auffallend an dieser Architektur sind Abweichungen von üblichen RISC-Architekturen, die aus der engen Verbindung zwischen Compiler-Technologie und Rechnerarchitektur entstanden. Diese Verbindung äußert sich in Merkmalen wie dem *load/store*-Modus, der die Analyse der Wertübertragungsbeziehungen vereinfacht, Drei-Adreß-Befehle passend zur Compiler-Zwischensprache des Drei-Adreß-Codes und einer großen Zahl von allgemein verwendbaren Registern (meistens 32). Programme in imperativen Sprachen mit relativ großen Sequenzen

[2]Tycoon Register Machine

von Anweisungen ohne Prozeduraufrufe können in sehr effizienten Maschinencode für RISCs übersetzt werden. Eine ausführliche Erläuterung dieser Zusammenhänge findet sich in [Hennessy, Patterson 90].

Für primitive Operationen ist die Zwischensprache TML isomorph zu Drei-Adreß-Code, aber größere Sequenzen von primitiven Anweisungen kommen seltener vor und werden vielfach von Prozeduraufrufen unterbrochen. Dadurch verringert sich die Zahl der Variablen, die über längere Zeit in Registern gehalten werden können, und der Vorteil von Drei-Adreß-Befehlen, durch die zusätzliche Register-Lade-Operationen vermieden werden können, reduziert sich erheblich. Daher ist die Entscheidung für ein Zwei-Adreß-Format mit weniger Register sinnvoll. Eine kompakte Bytecodierung der Befehle ist möglich, wenn in der Architektur maximal 16 Register vorgesehen sind, da nur vier Bits für die Codierung der Operanden benötigt werden.

7.2 Codegenerierung für die Registermaschine

Der CPS-Dialekt TML wird ohne weitere Transformationen[3] in Maschinencode übersetzt. Wie schon in Abschnitt 5.1 erwähnt, kann TML als Lambdakalkül-Dialekt in zwei syntaktische Kategorien (Applikationen und Abstraktionen) aufgeteilt werden, die bei der Codeerzeugung unterschiedlich behandelt werden.

7.2.1 Applikationen

Die applikative Natur von TML als strikter CPS-Dialekt bestimmt den Ablauf der Codegenerierung: Erst werden die Argumente einer Funktion (oder primitiven Operation) ausgewertet und in Zielregistern abgelegt und anschließend wird die Funktion (oder primitive Operation) auf die Argumente angewandt. In TML treten vier Formen von Applikationen auf:

1. Lambda-Applikationen
2. Funktionsapplikationen
3. Anwendung primitiver Operationen
4. Fortsetzungsapplikationen

Der folgende TML-Ausschnitt enthält alle vier Applikationsarten: a wird durch eine Lambda-Applikation an 3 gebunden, die Addition ist ein TML-Primitiv, und das Ergebnis der Addition wird durch eine Fortsetzungsapplikation weitergereicht an die Funktionsapplikation (f 2 t_1 ...).

```
(lambda(a)
  . (+ a 5 cont(t_1)
  .    (f 2 t_1 cc))
  . 3)
```

[3] Im Rückblick hat sich diese Tatsache als nachteilig erwiesen, worauf in Abschnitt 7.2.2 noch eingegangen wird.

Die Auswertung der Argumente und die Code-Emission erfolgen je nach Applikationsart unterschiedlich:

1. Lambda-Applikationen dienen dazu, lokale Variablen zu binden, wobei die Argumente in beliebige Register abgelegt werden. Für die Lambda-Applikation im obigen Beispiel könnte zum Beispiel durch die Emission von „`ld_int R1 3`“ der Wert 3 an `a` gebunden werden[4].

2. Bei Funktionsapplikationen müssen die Parameterübergabekonventionen der TRM eingehalten werden, d.h. die Argumente werden in den Registern R0...R15 abgelegt.

3. Die Anwendung primitiver Operationen erfordert keine vorgeschriebenen Register und bietet eine Möglichkeit, die Qualität des Codes durch Registerallokation zu verbessern.

4. Fortsetzungen erhalten ihr Argument aus der vorherigen Applikation, und erweitern die lokalen Bindungen um ihren Parameter. Wenn die vorherige Applikation ein Prozeduraufruf war, liegt das Argument in R0 vor (bedingt durch die Aufrufkonventionen der TRM), sonst in einem beliebigen Register. Fortsetzungsaufrufe werden auf drei Arten übersetzt:

   a) Aufrufe an ungebundene Lambda-Fortsetzungen von Applikationen erfordern keinen Code.

   b) Gebundene Fortsetzungen werden in einen Sprung übersetzt.

   c) Für den Aufruf der Rücksprungfortsetzung wird ein `return`-Bytecode emittiert.

Unter der Annahme, daß die Funktion `f` aus dem Beispiel an der ersten Stelle im aktuellen Umgebungsvektor zu finden ist, und dieser in R15 liegt, ist eine geeignete Codesequenz für das Beispiel:

```
ld_int    r1   3     ; Binden von 3 an  a          (lambda(a) ...) 3
addi      r1   5     ; Primitive Operation         (+ a 5 ...
mov       r2   r1    ; Erweitern der Bindungen     cont(t_1)
mov       r0   r15   ; Umgebung nach r0            (f 2 t_1 ...)_1
geti      r0   1     ;  f  nach r0                 (f 2 t_1 ...)_2
ld_int    r1   2     ; 2 nach r1                   (f 2 t_1 ...)_3
call                 ; Funktionsaufruf             (f 2 t_1 ...)_4
return    0          ; Rücksprungfortsetzung       cc
```

7.2.2 Abstraktionen

Abstraktionen als gleichberechtigte Werte in TML erfordern eine besondere Behandlung bei der Codegenerierung, um die Semantik der lexikalischen Sichtbarkeitsregeln und die Eigenschaft von Funktionen als gleichberechtigte Werte zu realisieren. Ein Funktionsabschluß muß einen Teil der Erzeugungsumgebung, nämlich die Menge der freien Bindungen, beinhalten, damit unabhängig von der aktuellen Ausführungsumgebung alle referenzierten Werte erreichbar

[4] Die Strategie zur Auswahl der Register wird in Abschnitt 7.3.2 erläutert; die Wahl `R1` hat an dieser Stelle keine besondere Bedeutung.

bleiben (selbst wenn die Erzeugungsumgebung nicht mehr existiert). In dem Tycoon-System müssen Funktionsabschlüsse zusätzlich speicherbar sein.

Die folgenden Schritte aus dem TVM-Codegenerator zur Übersetzung von Funktionsabschlüssen (*closure conversion*) werden auf TRM-Mechanismen abgebildet:

1. Der Rumpf einer Abstraktion wird in Bytecode übersetzt, wobei die freien Variablen der Abstraktion in einer Liste eingesammelt werden. Die Lage einer Variable in dieser Liste entspricht dem Index, an dem diese Bindung im Umgebungsvektor liegen wird. Dieser Index ist daher schon während der Code-Emission des Rumpfs bekannt.

2. Der übersetzte Bytecode wird mit den Literalen des Rumpfes während der Übersetzung in einen Literalvektor zusammengefaßt, der wiederum zu den Literalen des umgebenden Kontextes hinzugefügt wird und somit im umgebenden Kontext referenzierbar ist. Eine verkettete Struktur (siehe Abbildung 7.3) von Literalvektoren entsteht, die im Literalvektor der obersten Funktion eingetragen wird.

3. Für den umgebenden Ausführungskontext der Abstraktion wird Code generiert, der den Umgebungsvektor anlegt und die freien Bindungen einträgt. Dieser Code enthält außerdem Befehle, die den Literalvektor des Rumpfes (erreichbar über die Literale des umgebenden Kontextes) im Umgebungsvektor des Funktionsabschlusses eingetragen.

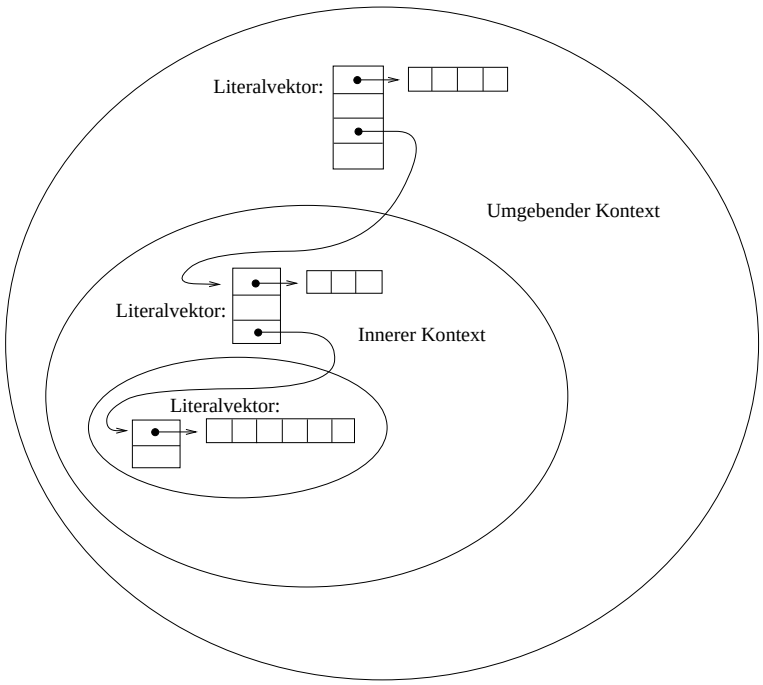


Abbildung 7.3: Verkettung von Literalvektoren

Das Vorgehen soll an folgender Beispielabstraktion erläutert werden:

```
proc(cc)
  (cc a)
```

Diese Abstraktion besitzt nur eine freie Variable `a`, daher wird `a` an der Position mit dem Index 1 im Umgebungsvektor[5] liegen, woraus sich der Index des `geti`-Befehls im Bytecode des Rumpfes ergibt:

```
geti r0 1    ; erste globale Variable im Umgebungsvektor: a
return  0    ; Rückgabe über r0
```

Der Code zum Anlegen des Funktionsabschlusses wird im umgebenden Kontext der Abstraktion ausgeführt. Unter der Annahme, daß im umgebenden Kontext `a` lokal in R3 vorliegt und der Literalvektor der Abstraktion **proc**(cc)(cc a) sich an dem Index 2 im aktuellen Literalvektor befindet, sieht der Bytecode zum Anlegen des Funktionsabschlusses wie folgt aus:

```
closure r0 2      ; Allokation von 2 Worten für  a und den Literalvektor
seti  1 r0 r3     ; Eintragen von  a
ld_lit  r15 2     ; Literalvektor des Rumpfes nach r15
seti 0  r0 r15    ; Eintragen des Literalvektors
```

Abbildung 7.4 deutet an, wie diese Codefragmente in Literalvektoren verkettet werden und auf welche Weise der Bytecode eines anzulegenden Funktionsabschlusses zur Laufzeit im aktuellen Literalvektor erreichbar ist.

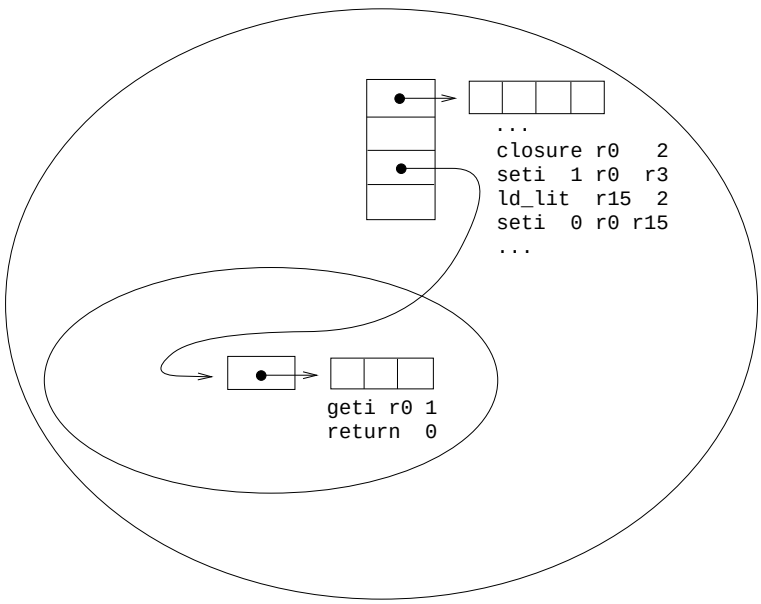


Abbildung 7.4: Literalvektoren für die Abstraktion **proc**(cc)(cc a)

Eine Erkenntnis aus der Entwicklung des Codegenerators betrifft die Übersetzung von Abstraktionen: in TML als Sprache höherer Ordnung sind Abstraktionen Objekte erster Klasse

[5] Der Umgebungsvektor liegt per Konvention bei einem Funktionsaufruf in R0.

wie in TL oder TooL. Wie an dem vorherigen Beispiel erkennbar ist, hat die Erzeugung von Funktionsabschlüssen in Maschinencodeschritten keine Entsprechung im TML-Code. In dem in dieser Arbeit verwendeten Registerallokationsverfahren (siehe Abschnitt 7.3) wirkt sich diese Tatsache nachteilig aus, da die Registerallokation auf der TML-Repräsentation arbeitet, und die Erzeugung von Funktionsabschlüssen zu nicht explizit erkennbaren Registeranforderungen führt. Eine Lösung ist, ähnlich wie in dem ML-Übersetzer von Appel [Appel 89] oder dem CPS-Backend von Kelsey und Hudak [Kelsey, Hudak 89] obige Schritte als zusätzliche TML-Transformationsphase durchzuführen, und damit alle Variablenbenutzungen und Registeranforderungen offenzulegen.

Wechselseitig rekursive Abstraktionen

Bei wechselseitig rekursiven Abstraktionen tritt das Problem auf, daß einige der freien Variablen einer Abstraktion zum Zeitpunkt des Anlegens des Funktionsabschlusses noch nicht gebunden sind, und zwar die jeweils wechselseitig rekursiven Bezeichner. Der folgende TML-Ausschnitt verdeutlicht diesen Sachverhalt:

```
(Y proc(c_1 g f)
(c_1
  cont()
    (c g)
  proc(c_2)          ; Abstraktion  g
    (c_2 f)          ; besitzt  f als freie Variable
  proc(c_3)          ; Abstraktion  f
    (c_3 g)))        ; besitzt  g als freie Variable
```

Wenn der Funktionsabschluß von g angelegt wird, ist die freie Variable f im Umgebungskontext noch nicht gebunden. Sie wird erst nach Anlegen des Funktionsabschlusses von f, also nach der Übersetzung der Abstraktion f gebunden. Daher werden die Einträge der ungebundenen freien Variablen in den Globalvektoren erst nach der Übersetzung aller wechselseitig rekursiven Abstraktionen vorgenommen. Für obigen TML-Code sieht der emittierte Maschinencode folgendermaßen aus:

```
closure   r15     2     ; Anlegen des Funktionsabschlusses  g
ld_lit    r14     1     ; Literalvektor von  g an Index 1 im aktuellen Kontext
seti  0   r15     r14   ; Eintrag des Literalvektors

closure   r14     2     ; Anlegen des Funktionsabschlusses  f
seti  1   r14     r15   ;  g ist schon gebunden
ld_lit    r13     2     ; Literalvektor von  f an Index 2 im aktuellen Kontext
seti  0   r14     r13   ; Eintrag des Literalvektors

seti  1   r15     r14   ; Nachtrag:  f in  g eintragen
```

7.3 Registerallokation

Ein wesentlicher Aspekt bei der Codegenerierung für Registerarchitekturen ist eine Registerallokation. Eine geschickte Verwaltung der knappen Ressource „Register“ führt zu effizientem und elegantem Maschinencode, und auf realen Maschinen werden Variablenzugriffe

entscheidend beschleunigt. Das verbreitete Verfahren, die Allokation als Färbungsproblem aus der Graphentheorie zu modellieren [Chaitin et al. 81] [Chow, Hennessy 90], läßt sehr effiziente Registerbelegungen zu, wird aber nicht für CPS-basierte Übersetzer eingesetzt. Durch die hohe Dichte an Prozeduraufrufen sind Grundblöcke kürzer als in imperativen Sprachen, so daß die Menge der lebendigen Variablen in der Regel klein ist. Ferner diktieren die Parameterübergabekonventionen einen Großteil der Registerbelegungen, so daß die Allokationsfreiheit zusätzlich eingeschränkt ist. Anstelle einer umfassenden Allokation wird in dem TRM-Codegenerator eine Registerzuordnung (*register targeting*) nach [Wulf et al. 73] vorgenommen, eine Methode, die auch in ML-Übersetzern [Appel 89] und Lisp-Übersetzern [Teodosiu 91] [Kranz et al. 86] zum Einsatz kommt. Die Registerzuordnung besteht aus zwei getrennten Phasen: einer Lebendigkeitsanalyse und einem Zuordnungslauf.

7.3.1 Lebendigkeitsanalyse

In [Wulf et al. 73] wird ein Näherungsverfahren für die Berechnung von Lebendigkeitsbereichen von Bezeichnern beschrieben, das auch in dem TRM-Codegenerator eingesetzt wird. Jeder Knoten eines Programms erhält eine Ordnungsnummer, welche die Position des Knotens als Tupel von zwei Zahlen angibt: der Linearordnungszahl und der Verzweigungsordnungszahl. Die Linearordnungszahl entspricht etwa einer „Zeilennummer“ , die sich aus einer Linearisierung des Programms ergäbe (verschiedene Blöcke eines Konditionals werden in Maschinenanweisungen übersetzt, die hintereinander im Speicher liegen). Die Linearordnungszahl wird von einem Knoten zum nächsten inkrementiert. Die Verzweigungsordnungszahl ist für verschiedene Arme einer Verzweigung gleich und wird entlang eines Arms ebenfalls inkrementiert. Die Verzweigungsordnung ermöglicht einen einfachen Test, ob verschiedene Variablen in den gleichen Zweigen lebendig sind.

Die Berechnung der Ordnungstupel erfolgt in einem einfachen Durchlauf des TML-Baums. Abbildung 7.5 zeigt die Tupel <Linearordnung,Verzweigungsordnung> für einen TML-Ausschnitt.

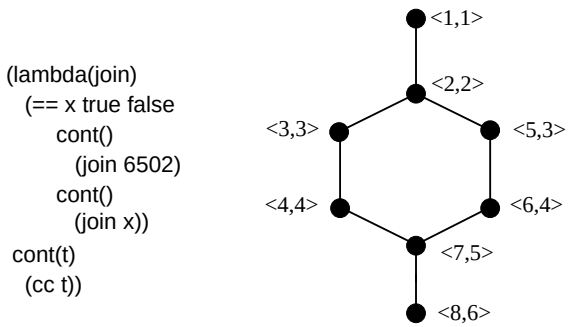


Abbildung 7.5: Linearordnung und Verzweigungsordnung eines TML-Baums

Die Zuordnung der Ordnungszahlen auf das Bespiel in Abbildung 7.5 ist folgendermaßen:

```
(lambda(join)          < 1, 1 >
 . (== x true false    < 2, 2 >
 .  cont()             < 3, 3 >
 .    (join 6502)      < 4, 4 >
 .  cont()             < 5, 3 >
 .    (join x))        < 6, 4 >
  cont(t)              < 7, 5 >
    (cc t))            < 8, 6 >
```

Der Lebendigkeitsbereich eines Bezeichners ist durch die Ordnungsposition seiner ersten Definition und die Position seiner letzten Nutzung gegeben. Globale Bezeichner erhalten die Position des Funktionskopfes als Definitionsposition. In dem Beispiel besitzt x den Lebendigkeitsbereich <1,1>-<6,4> und t den Lebendigkeitsbereich <7,5>-<8,6>. Derartig definierte Lebendigkeitsbereiche lassen sich als Rechtecke auffassen (siehe Abbildung 7.6), deren linker unterer Eckpunkt die Definitionsposition ist und deren rechter oberer Eckpunkt durch die Position der letzten Nutzung gebildet wird. Ein Lebendigkeitskonflikt zweier Bezeichner ist an der Überschneidung der jeweiligen Rechtecke erkennbar.

Die Berechnung der Lebendigkeitsbereiche erfolgt in einem zweiten Baumdurchlauf, in dem die Definitionspunkte und letzten Nutzungspunkte für alle Bezeichner in eine Tabelle eingetragen werden.

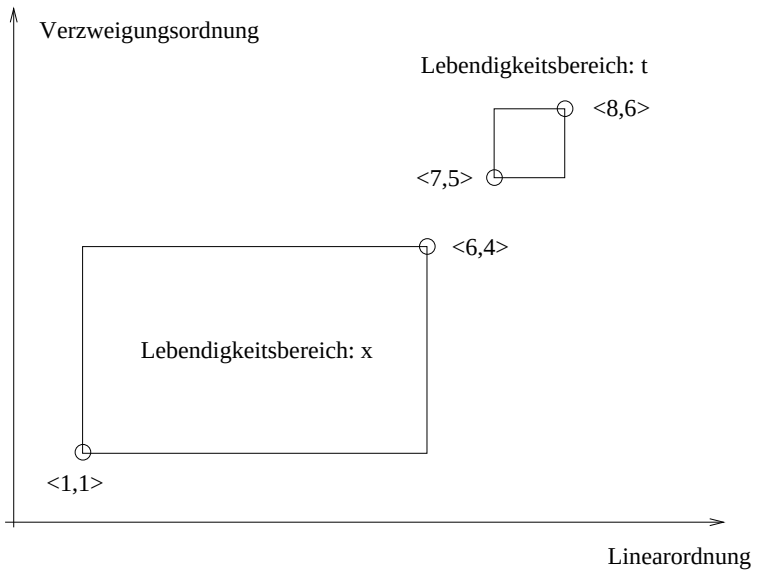


Abbildung 7.6: Lebendigkeitsbereiche als Rechtecke

7.3.2 Registerzuordnung

In dem Registerzuordnungsverfahren können Registerpräferenzen für lokale Variablen ermittelt werden. Allerdings werden im Gegensatz zu dem in [Wulf et al. 73] beschriebenen Verfahren im Tycoon-Codegenerator ähnlich wie bei [Appel 89] Zielregister (*targets*) bei einem

Baumaufstieg und nicht bei dem Abstieg abgeleitet. Dabei lassen sich aus vorgeschriebenen Registerbelegungen, die aus den Prozeduraufrufkonventionen und den Beschränkungen einiger Befehle folgen, geeignete Register für frühere Vorkommen einer Variable finden, so daß unnötige Lade-Befehle vermieden werden können. Das einfache Beispiel in Abbildung 7.7 veranschaulicht diesen Zusammenhang.

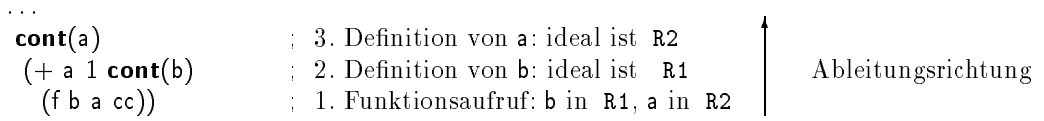


Abbildung 7.7: Zuordnungsbeispiel

Der im TRM-Codegenerator eingesetzte Zuordnungsalgorithmus nutzt sowohl den Baumabstieg als auch den Aufstieg:

- Der Aufstieg dient der Ableitung geeigneter Definitionsregister und Nutzungsregister für Operanden primitiver Operationen. Trifft man beim Aufstieg auf eine vorgeschriebene Nutzung eines Registers, so wird diese weiter nach oben propagiert. In dem Beispiel 7.7 erfordert der Aufruf von f, daß b in R1 und a in R2 vorliegt. Bei der Anwendung der primitiven Operation „+“ weiter oben im Baum ist für a also R2 ein günstiges Nutzungsregister, und die Definition von a sollte geeigneterweise auch R2 verwenden.

- Der Abstieg wird dazu genutzt, Kandidaten für eine Keller-Allokation zu ermitteln. Im Keller müssen alle Variablen alloziert werden, deren Lebendigkeitsbereich mehrere Grundblöcke überspannt. Das Ende eines Grundblocks ist durch TML-Konstrukte mit mehr als einer Fortsetzung (z.B. Konditionale), Funktionsaufrufe und den Fixpunkt-Kombinator Y gekennzeichnet.

  Das Verfahren beim Abstieg ist folgendermaßen: Alle Definitionen von lokalen Variablen werden akkumulierend weitergereicht, bis der Abstieg auf einen der begrenzenden Knotentypen stößt. Alle akkumulierten Variablen, die mit ihrem Lebendigkeitsbereich diesen Knoten überspannen, werden zur Allokation in den Keller vermerkt.

7.3.3 Zuteilung der Register

Die endgültige Zuteilung der Register erfolgt während der Codegenerierung. Hierzu werden in Anlehnung an [Aho et al. 86] zwei Tabellen geführt: eine für den aktuellen Inhalt der Registerbank und eine Tabelle für die Aufenthaltsorte aller Variablen. Die Registervergabe erfolgt durch eine zentrale Funktion *resolveLocation*, die ein Datum (Variable, Literal, oder Abstraktion) und ein Zielregister als Parameter erhält, und je nach Inhalt der Tabellen und Lebendigkeitsinformation wie folgt agiert:

1. Liegt das Datum schon in dem gewünschten Register, ist keine Codeemission erforderlich.

2. Liegt das Datum in einem anderen Register `Rx`, so lautet die nötige TRM-Anweisung:
   ```
   mov Rziel Rx
   ```

3. Variablen, die sich im Keller befinden, werden von dem Keller in das Zielregister befördert:
   ```
   ld_sp Rziel offset
   ```

4. Globale Variablen müssen über den Globalvektor geholt werden (der vorher durch eine rekursive Anwendung der Funktion *resolveLocation* in ein weiteres Register gebracht werden muß):
   ```
   mov Rziel Rglobalvektor
   geti Rziel offset
   ```

5. Literale werden aus dem Literalvektor gelesen:
   ```
   ld_lit Rziel offset
   ```

6. Sollten Konflikte bei der Zuteilung auftreten, weil das Zielregister ein lebendiges Datum enthält, wird der Inhalt des Zielregisters im Keller ausgelagert.

7. Falls das Zielregister nicht spezifiziert ist, wird das nächste freie Register ohne lebendigen Inhalt vergeben.

8. Alle Änderungen der Aufenthaltsorte werden in den Tabellen vermerkt.

Die Wirksamkeit dieses Registerallokationsverfahrens soll an dem Beispiel der Fakultätfunktion demonstriert werden. Von dem folgenden TML-Code, der wie der TRM-Code von der Ausgabe des Übersetzers übernommen wurde, sind aus Gründen der Übersichtlichkeit einige Teile (wie die Ausnahmebehandlung) weggelassen; an den Ergebnissen der Registerzuteilung für den TRM-Code ändert dieses nichts:

```
proc(n c_1)
 (== n 1
    cont()
     (c_1 1)
    cont()
     (- n 1 cont(t_1)
     (f t_1 cont(t_2)
     (* n t_2  c_1))))
```

Der dazugehörige TRM-Code ist:

```
    push        r0              ; f   (1)
    push        r1              ; n   (2)
    eqi         r1      1       ;
    branch_cf           l0      ;
    branch              l1      ;
l0:
    ld_int      r0      1       ;
    return      2               ;     (3)
l1:
    ld_sp       r1      0       ; n
    subi        r1      1       ;     (4)
    ld_sp       r0      1       ; f
    call        1               ; f
    move        r2      r0      ; t
    ld_sp       r0      0       ; n
    mul         r0      r2      ;     (5)
    return      2               ;
```

Zu einigen Punkten des TRM-Codes läßt sich folgendes anmerken:

1. Die Funktion f muß als rekursive Funktion im Keller abgelegt werden.
2. Der Lebendigkeitsbereich des Parameters n überspannt unter anderem den Funktionsaufruf (f t_1 ...) und muß daher auch im Keller liegen.
3. Die beiden Kellereinträge müssen vor dem Rücksprung aus dem Keller gelöscht werden.
4. Da das Ergebnis von (- n 1 ...) als erster Parameter des rekursiven Aufrufs von f in dem Register R1 erwartet wird, schlägt der Zuordnungsalgorithmus das Register R1 für die Definition von t_1 vor.
5. Das Register R0 wird als Ergebnisregister des letzten Schritts (* n t_2 ...) gewählt, da Funktionen ihre Ergebnisse in R0 ablegen.

Anhand der gemessenen Laufzeiten für einige Benchmarks erfolgt in dem nächsten Kapitel ein quantitativer Vergleich mit der Kellermaschine. Vorab läßt sich feststellen, daß das Codegenerierungsverfahren für die TRM erheblich aufwendiger als die Codegenerierung für die TVM ist, und daß der Performanzgewinn zu gering ist, um die Registermaschine als Ausführungsplattform zu rechtfertigen.

8. Quantitative Ergebnisse

Ein wichtiger Bestandteil einer Arbeit mit dem Ziel einer Optimierung oder Effizienzsteigerung ist die Durchführung von Leistungsmessungen, denn diese fungieren mangels formaler Beweise als Anhaltspunkt für die Bestätigung oder Widerlegung von Hypothesen wie „Durch den Einsatz von markierten Typen, Laufzeitprofiling und Strategien zur Elimination von Endrekursion können die Effizienznachteile von rein objektorientierten Sprachen deutlich vermindert werden“, oder „Eine virtuelle Registermaschine ist eine geeignete Ausführungsplattform und führt zu geringeren Laufzeiten“. In diesem Kapitel werden zwei Fragestellungen mit Messungen untersucht: Wie entscheidend können die Nachteile von rein objektorientierten Sprachen vermindert werden? Wie effizient ist eine virtuelle Registerarchitektur für Sprachen wie TL oder TooL?

Die Metrik für Effektivität werden Ausführungszeiten verschiedener Testprogramme sein, die mit und ohne Optimierungstechnik übersetzt werden. In diesem Zusammenhang ist es hilfreich, die Begriffe Performanz und Performanzsteigerung (*speedup*) nach [Hennessy, Patterson 90] zu präzisieren. Die Performanz ist die Ausführungsgeschwindigkeit eines Programms als Kehrwert der Ausführungszeit:

$$Performanz = \frac{1}{Ausführungszeit}$$

Die Performanzsteigerung einer Optimierung folgt aus der Feststellung, daß Programme durch Übersetzung mit einer Optimierungstechnik schneller ablaufen als solche, die ohne Optimierungstechnik übersetzt werden:

$$Performanzsteigerung = \frac{Performanz_{optimiert}}{Performanz_{nicht\ optimiert}}$$

Im folgenden werden anhand der Ausführungszeiten von Testprogrammen die Performanzsteigerungen durch den TooL-Optimierer einerseits und den Einsatz der virtuellen Registermaschine andererseits ermittelt und diskutiert.

8.1 Die Performanz von TooL

Ziel der Meßreihen dieses Abschnitts ist nicht nur, die Auswirkungen der Optimierungen festzuhalten, sondern auch ansatzweise Aufschluß darüber zu geben, wie die Performanz von TooL im Vergleich zu anderen Programmiersprachen steht.

Die Auswahl des Testsatzes ist sowohl von pragmatischen Gedanken als auch von Gesichtspunkten der Aussagefähigkeit geleitet. Grundsätzlich gelten die Laufzeiten von größeren Programmen und echten Anwendungen als aussagekräftiger als die von kurzen Testprogrammen. Da zum Zeitpunkt dieser Arbeit kaum reale TooL-Anwendungen existieren, bilden künstliche Benchmarks aus der Stanford-Suite den Grundstock der Testprogramme. Der Vorteil dieser kurzen Testprogramme liegt in der allgemeinen Verfügbarkeit, so daß Vergleiche zwischen Programmiersprachen eventuell möglich sind. Als Ersatz für reale Anwendungen dienen der Richards-Benchmark, eine Simulation von Betriebssystemmechanismen mit Funktionseinheiten, Warteschlangen, Aufträgen und einem rudimentären Scheduler, und eine Implementierung eines generischen n-dimensionalen Wörterbuchs [Römer, Lotter 93]. Durch die Wiederverwendung von Code in objektorientierten Systemen läßt sich die Größe dieser Programme kaum abschätzen, was vor allem für TooL als rein objektorientierte Sprache gilt. Die von C++ nach TooL portierte Version des Richards-Benchmarks umfaßt circa 900 Zeilen TooL-Quelltext, das n-dimensionale Wörterbuch ist circa 2500 Zeilen lang. Beide Programme machen ausgiebig Gebrauch von verschiedenen Behälterklassen aus der TooL-Klassenbibliothek, so daß deutlich mehr Code zur Ausführung kommt, und diese Programme als Testfälle mittlerer Größe gelten können.

8.1.1 Performanzsteigerungen durch Optimierungen

Um die möglichen Performanzsteigerungen durch das in dieser Arbeit beschriebene Optimierungsverfahren zu messen, werden für jedes Testprogramm die Ausführungszeiten vor und nach der Optimierung gemessen. Die Testfälle sind:

1. Quicksort von 10000 zufälligen Elementen
2. Bubblesort von 1000 zufälligen Elementen
3. Treesort von 10000 zufälligen Elementen
4. Queens (das Acht-Damen-Problem)
5. Permutations (stark rekursives Testprogramm aus der Stanford-Suite)
6. Puzzle (Testprogramm aus der Stanford-Suite)
7. Richards
8. nDimDict (Erzeugung eines n-dimensionalen Wörterbuchs)

Nur die Programme Queens, Puzzle, Permutations und Richards erlauben eventuell einen Vergleich mit anderen Programmiersprachen, da sie explizit als Benchmarks in TooL codiert sind. Die Sortierverfahren entstammen der TooL-Behälterbibliothek. Daher sind sie aus Sicherheitserwägungen für Bibliothekscode mit Zusicherungscode und zusätzlichen Konsistenzabfragen erweitert, die einen direkten Vergleich mit den einfach codierten Routinen aus der ursprünglichen Stanford-Suite unmöglich machen. Die Messungen der Laufzeiten der TooL-Programme verliefen unter Einsatz der virtuellen Kellermaschine TVM. Die Testläufe fanden auf einer unbelasteten SparcStation 20 von SUN Microsystems mit 2 Prozessoren und 256 MB Hauptspeicher statt. Die berichteten Zeiten sind CPU-Zeiten.

Benchmark	*Zeit: TooL (s)*	*Zeit: TooL optimiert (s)*	*Performanzsteigerung*
quicksort	44.85	24.92	1.79
bubblesort	133.1	67.63	1.97
treesort	41.07	16.11	2.55
perms	8.74	3.26	2.68
queens	9.57	4.7	2.04
puzzle	71.36	32.12	2.22
richards	216.86	69.09	3.14
nDimDict	49.76	16.37	3.04

Abbildung 8.1: Performanzsteigerungen durch Optimierung von TooL-Programmen

Die gemessenen Zeiten der Testläufe und die Performanzsteigerungen sind in Abbildung 8.1 festgehalten. In den nicht-optimierten Programmen wird jede Operation im Quelltext durch einen Methodenaufruf realisiert. Die aufgefundenen Implementierungen eines Methodenaufrufs werden allerdings in einem zentralen Methodencache gehalten, um den Aufwand des Aufrufs möglichst gering zu halten. Bei allen Läufen war der Profiler zugeschaltet. Das Ergebnis dieser Messungen ist: TooL-Programme, die mit den in dieser Arbeit beschriebenen Techniken optimiert werden, können zwei bis drei mal schneller ablaufen als solche, die ohne Optimierung übersetzt werden. Ganz überraschend sind diese hohen Beschleunigungsfaktoren nicht, denn schon die Ersetzung der rein objektorientierten Kontrollstrukturen durch einfache Schleifen im Rahmen der Optimierungen dürfte einen großen Teil der Performanzsteigerungen ausmachen.

Benchmark	*TooL (Botschaften/s)*	*TooL optimiert (Botschaften/s)*	*TooL/TooL-opt*
quicksort	136235.65	119818.58	1.14
bubblesort	140902.71	103380.85	1.36
treesort	112498.98	58995.22	1.91
perms	149084.10	83296.93	1.79
queens	137869.49	89076.38	1.55
puzzle	128011.27	84939.07	1.51
richards	147831.78	97438.65	1.52
DDimDict	155553.05	82204.95	1.89

Abbildung 8.2: Methodenaufrufhäufigkeiten

Es stellt sich die Frage, ob nicht noch bessere Ergebnisse erzielbar sind. Hierzu soll der dynamische Anteil an Methodenaufrufen vor und nach einer Optimierung untersucht werden. Für jedes Testprogramm wird mit Hilfe eines Aufrufzählers die Zahl der Methodenaufrufe festgehalten, die anschließend durch die Laufzeit des Programms geteilt wird, um eine Methodenaufrufhäufigkeit zu erhalten. Diese Frequenzen sind in Abbildung 8.2 dargestellt.

Es ist erkennbar, daß die dynamische Häufigkeit von Methodenaufrufen durch die Optimierungen im Durchschnitt um einen Faktor von 1,6 verringert wird. Hölzle berichtet, daß der SELF-Übersetzer (SELF93) durch Profiling mit Hilfe eines polymorphen Inline-Caches und erneuter Übersetzung die Botschaftsfrequenz um einen Faktor von 3,6 reduziert [Hölzle 94], also ein deutlich besseres Ergebnis als das des TooL-Optimierers.

Ein Grund für diesen Unterschied kann darin liegen, daß der TooL-Optimierer keine Klassenabschätzung durchführt, während das polymorphe Inline-Caching von SELF als eine generalisierte Klassenabschätzung verstanden werden kann. Dadurch entgehen dem TooL-Optimierer einige Möglichkeiten, Methodenaufrufe durch Funktionsaufrufe zu ersetzen, und es treten noch Botschaften wie „+,*,-“ usw. auf. Diese könnten durch selektorbasierte Klassenabschätzung eliminiert werden, allerdings wäre dann eventuell auch noch eine Splitting-Phase im Optimierer notwendig. Alternativ dazu ließe sich das Konzept der markierten Typen, das sich als eine der Hauptinformationsquellen für den Optimierer herausgestellt hat, in den Typinferenzmechanismus des Typprüfungsalgorithmus einführen, so daß zum Zeitpunkt der Generierung des TML-Zwischencodes die Signaturen von allen Bezeichnern offenliegen. So können schon bei der Transformation von TooL-AST nach TML-AST einige Methodenaufrufe an Methoden der markierten Typen zum Übersetzungszeitpunkt eliminiert werden.

8.1.2 Codezuwachs durch Optimierungen

Durch die Optimierungsmechanismen dieser Arbeit wird der Code in der Regel größer, da Methoden an ihren Aufruforten integriert und Hilfsfunktionen wie while teilweise mehrfach angelegt werden (wenn diese sich nicht integrieren lassen). Die Codegröße läßt sich folgendermaßen messen: Nach dem Lauf eines nicht-optimierten Programms kann der Cache unter Berücksichtigung aller in einem Funktionsabschluß gehaltenen Funktionen traversiert und die gesamte Codegröße des Programms (in Bytes) ermittelt werden. Anschließend wird die Optimierung gestartet und nach einem erneuten Testlauf (um den Cache mit Methoden, die bei $\theta = 0.001$ nicht optimiert werden, zu füllen) kann der Cache wiederholt traversiert werden. Die Ergebnisse dieser Messungen sind in Abbildung 8.3 dargestellt.

Der Code wird im Durchschnitt um den Faktor 1.76 größer, ein Wert, der das in Abschnitt 6.4.4 erwähnte Problem der redundant gehaltenen Hilfsfunktionen eines Funktionsabschlusses relativiert.

Messungen der Codegröße bei $\theta = 0$ zeigen, daß in diesem Fall der Code im Durchschnitt um den Faktor 3.75 wächst. Der heuristisch gewählte Wert von $\theta = 0.001$, bei dem 90 % bis 95 % der Performanzsteigerungen erreicht werden, scheint sich auch bezüglich der Codegröße zu bewähren.

Benchmark	*Codegröße: TooL (bytes)*	*Codegröße: TooL-opt*	*TooL-opt/TooL*
quicksort	1139	2401	2.11
bubblesort	1146	2404	2.10
treesort	1217	1387	1.14
perms	1219	1389	1.14
queens	1384	2357	1.70
puzzle	3045	3795	1.25
richards	8495	16985	2.00
DDimDict	10232	26697	2.61

Abbildung 8.3: Codezuwachs durch Optimierungen

8.1.3 TooL im Vergleich zu TL und Java

Die Frage nach einem Leistungsvergleich zwischen Programmiersprachen und Übersetzern stellt sich immer wieder und läßt sich in der Regel nicht einfach durch Messungen beantworten. Besonders schwierig ist der Vergleich zwischen semantisch mächtigen Sprachen wie TooL und gängigen, einfacheren Sprachen wie C oder C++. TooL besitzt Eigenschaften wie reine Objektorientierung, inkrementelle Erweiterung des Klassengraphen, orthogonaler Persistenz und ein portables Ausführungsformat, die in C oder C++ nicht unterstützt werden. Dennoch soll im folgenden nur die Ausführungsgeschwindigkeit als Vergleichsgröße gelten, um einen Eindruck der absoluten Leistung von TooL-Programmen zu erhalten.

TooL vs. TL

Eine ideale Sprache als Vergleichskandidat ist TL, denn TL und TooL besitzen einen gemeinsamen architektonischen Ursprung: Zwischencode, Codegenerator, Bytecode, Interpreter und Laufzeitsystem sind bis auf Erweiterungen für den Methodenaufruf identisch. So kann die Performanz von TL als Meßlatte für TooL dienen. Von den Testprogrammen des letzten Abschnitts sind allerdings nur die explizit als Benchmarks codierten Programme sinnvoll. Schon eine zusätzliche Konsistenzabfrage in einer inneren Schleife eines Sortierprogramms würde zu irreführenden Aussagen verleiten, so daß kein Vergleich zwischen den Stanford-Sortierverfahren in TL und den Sortierverfahren aus der TooL-Klassenbibliothek möglich ist.

Benchmark	*Zeit: TL optimiert (s)*	*Zeit: TooL optimiert (s)*	*Performanzvorteil: TL/TooL*
perms	1.044	3.26	3.12
queens	1.856	4.7	2.53
puzzle	10.98	32.12	2.93
richards	11.05	69.09	6.25

Abbildung 8.4: Performanzvergleich zwischen TooL und TL

Die Laufzeiten der optimierten TL-Benchmarks sind in Abbildung 8.4 den Laufzeiten der optimierten TooL-Benchmarks gegenübergestellt. Diese Zahlen zeigen auf, daß TooL-Programme zwischen 2.5 und 6.3-mal langsamer sein können als vergleichbare TL-Programme. Diese Werte geben nur einen ersten Eindruck der Geschwindigkeit von TooL-Anwendungen, denn es ist zu erwarten, daß Programme, die ausgiebig Gebrauch der Klassenbibliothek machen, eher langsamer ablaufen. Für die TooL-Version des Richards-Benchmarks ist dies der Fall, denn dieser Benchmark ist im TooL-Stil geschrieben und trägt so typischere TooL-Merkmale als z.B. das Acht-Damen-Programm. Trotz dieses Vorbehalts sind diese Ergebnisse gut, insbesondere da TL keine objektorientierte Sprache ist und alle Referenzen nach dem Binden fest verknüpft sind.

In [Hölzle 94] wird berichtet, daß SELF-Programme im besten Fall halb so schnell wie C++-Programme ablaufen können (was einem Faktor von 2 entspricht), aber in C++-Programmen treten durchaus noch dynamische Methodenaufrufe auf, so daß es denkbar ist, daß SELF im Vergleich zu einer nicht-objektorientierten Sprache ähnlich abschneidet wie TooL.

TooL vs. Java

Die Programmiersprache Java [Gosling, McGilton 95] besitzt einige Gemeinsamkeiten mit TooL und ist deshalb ein weiterer Vergleichskandidat. Java ist ebenfalls objektorientiert und wird auch auf einer virtuellen Maschine interpretiert. Allerdings ist sie nicht rein objektorientiert und auch nicht orthogonal persistent wie TooL. Für diesen Performanzvergleich werden nur ähnlich codierte Benchmarks herangezogen, und die Messungen sollen nur einen ersten Eindruck von dem Performanzverhältnis festhalten. Die Zahlen in Abbildung 8.5 sprechen ein deutliches Urteil: TooL-Programme laufen um eine Größenordnung langsamer als Java-Programme. Woher dieser große Unterschied kommt, läßt sich ohne einen genaueren Vergleich der Interpreter, der Laufzeitsysteme und der Spracheigenschaften nur grob erklären: Die stichprobenartigen Messungen dieses Kapitels ergeben, daß optimierte TL-Benchmarks 2 bis 3-mal langsamer als Java-Benchmarks ablaufen. In diesen Java-Benchmarks treten fast keine dynamischen Methodenaufrufe auf, so daß ihre Ablaufmuster näher an denen der TL-Programme liegen. Dieser Faktor von 2 bis 3 mag an den Kosten der Persistenz liegen, denn jedes Objektspeicherwort ist mit Tag-Bits versehen, welche die Zugriffe verteuern. Die TooL-Benchmarks sind um 3 bis 6-mal langsamer als TL-Benchmarks, was auf verbleibende Botschaften zurückzuführen ist. Eine einfache Kombination dieser Faktoren ergibt mögliche TooL-Java Verhältnisse von 6 bis 18, was den gemessenen Werten entspricht.

Benchmark	*Zeit: TooL optimiert (s)*	*Zeit: Java (s)*	*Performanzvorteil: Java/TooL*
perms	3.26	0.45	7.24
queens	4.7	0.21	22.38
puzzle	32.12	2.56	12.55

Abbildung 8.5: Performanzvergleich zwischen TooL und Java

8.1.4 Fazit

Der TooL-Optimierer kann die Laufzeit von TooL-Programmen entscheidend verringern. Allerdings zeigt der Vergleich mit den Sprachen TL und Java, daß noch Handlungsbedarf besteht:

Die allgemeine Klassenanalyse dieser Arbeit, die keine Sonderfälle berücksichtigt, reicht nicht aus, um alle Methodenaufrufe an Basismethoden zu eliminieren. Es ist nicht realistisch anzunehmen, daß ein gänzlich unparteiischer Mechanismus ausreicht, um auch die häufig vorkommenden Aufrufe der Botschaften „*,-,=“ usw. naheliegenderweise entfernen zu können. Als Alternative zur Klassenabschätzung kann zur Übersetzungszeit mit Hilfe der markierten Typen ein Satz von häufigen Basisselektoren während der Abbildung des TooL-ASTs auf den TML-AST expandiert werden. Dieses setzt voraus, daß die Typinferenz der Typüberprüfung immer vor der Übersetzung von TooL nach TML abläuft (zur Zeit wird die Typüberprüfung nur optional durchgeführt). Diese Expansion kann sich auf einen Satz von Primitiven beschränken und muß nicht unbedingt Kontrollstrukturen beinhalten. Kontrollstrukturen können durch allgemeinere Transformationen während einem Optimierungslauf in effizientere Formen überführt werden. Auf jeden Fall würde die begrenzte Ausnutzung der Typattribute im TooL-AST durch die Festlegung einiger Basistypen sich anbieten, da das Typsystem von TooL diese Optimierungshinweise bereitstellt.

8.2 Performanzvergleich zwischen der Registerarchitektur und der Kellerarchitektur

Vor einem eventuellen Einsatz der Tycoon Registermaschine TRM als TooL-Plattform soll die Frage geklärt werden, wie leistungsfähig der Registermaschinencode ist, denn nur ein deutlicher Geschwindigkeitsvorteil würde den Aufwand eines Wechsels auf die TRM mit vollständigem TL-*bootstrap* rechtfertigen.

In diesem Abschnitt wird die TRM mit der Kellermaschine TVM anhand einer kleinen Reihe von Benchmarks aus der Stanford-Suite verglichen. Diese Tests wurden in der Sprache TL durchgeführt, da die Entwicklung von TooL zeitgleich zur Entwicklung des TRM-Codegenerators erfolgte, und der TRM-Interpreter noch keinen Methodenaufruf unterstützt[1]. Auch diese Tests erfolgten auf einer unbelasteten SparcStation 20 mit 2 Prozessoren und 256 MB Hauptspeicher.

8.2.1 Auswirkung der verschiedenen Interpreterimplementierungen

Zunächst soll die Auswirkung der verschiedenen Interpreterimplementierungen betrachtet werden. Von Interesse ist, ob die Interpreter grundlegende Differenzen besitzen, die einen direkten Vergleich unmöglich machen. Da die Instruktionssätze sich sehr stark unterscheiden, wurden die Ausführungszeiten von einigen TL-Primitiven statt den Zeiten einzelner Bytecodebefehle gemessen. Diese TL-Primitive können als atomare Berechnungseinheiten verstanden werden, denn sie bilden den Basissatz von Operationen, aus denen TL-Programme sich zusammensetzen.

Das Ergebnis der Messung in Tabelle 8.6 läßt den Schluß zu, daß die Interpreter sehr ähnlich codiert sind und keine bemerkenswerten Unterschiede bestehen. Die TRM erscheint leicht schneller für die Ausführung einzelner Berechnungseinheiten.

TL-Berechnungseinheit	*TRM-Zeit (ms)*	*TVM-Zeit (ms)*
Binden eines Wertes	0.02	0.04
Anlegen eines Funktionsobjektes	0.04	0.04
Funktionsaufruf	0.04	0.06
Tupelerzeugung	0.06	0.06
Tupelreferenz	0.04	0.06
Konditional	0.02	0.04
Arrayerzeugung	0.04	0.04

Abbildung 8.6: Vergleich der Ausführungszeiten für TL-Grundoperationen

8.2.2 Benchmarks

Die Ausführungszeiten der nicht-optimierten Testfälle aus der Stanford-Suite sind in Tabelle 8.7 wiedergegeben. Es ist deutlich erkennbar, daß der Code der Registermaschine nicht

[1] Der send-Bytecode ist zwar im Instruktionssatz vorgesehen, dessen Ausführung aber nicht implementiert.

so schnell wie der Code der Kellermaschine ist; im Durchschnitt liegt die Leistung der Registermaschine nur bei 95 % der Performanz der Kellermaschine. Der Grund hierfür liegt in der Tatsache, daß in nicht-optimierten TL-Programmen auch primitive Operationen in Funktionsaufrufe übersetzt werden. Dadurch hat die Registerallokation kaum Handlungsspielraum, denn viele Variablen müssen aufgrund der Aufrufdichte im Keller abgelegt werden. Vor jeder Operation werden zusätzliche Befehle notwendig, die Werte von dem Keller in die für Funktionsaufrufe vorgeschriebenen Register bewegen, und somit fällt der erzeugte Maschinencode weniger kompakt aus als der Kellermaschinencode.

Benchmark	*TRM-Zeit (ms)*	*TVM-Zeit (ms)*	*Performanzverhältnis: TRM/TVM*
quicksort	6.98	6.68	0.96
bubblesort	13.14	13.50	1.03
treesort	2.70	2.70	1.00
queens	99.14	94.4000	0.95
permutations	3264.80	2754.70	0.84
towers	4140.00	3766.00	0.91

Abbildung 8.7: Vergleich der Ausführungszeiten der nicht-optimierten Benchmarkprogramme

Um die Auswirkung der Registerallokation zu messen, wurden diese Benchmarkprogramme zusätzlich durch Expansion (*inlining*) der primitiven Funktionen optimiert. In Tabelle 8.8 ist erkennbar, daß die Registermaschine leicht schneller als die Kellermaschine ist, aber im Durchschnitt nur um den Faktor 1,05.

Benchmark	*TRM-Zeit (ms)*	*TVM-Zeit (ms)*	*Performanzverhältnis: TRM/TVM*
quicksort	3.80	3.90	1.03
bubblesort	7.10	8.30	1.17
treesort	1.90	1.90	1.00
queens	61.00	63.80	1.05
permutations	1285.00	1407.00	1.09
towers	2785.00	2660.00	0.96

Abbildung 8.8: Vergleich der Ausführungszeiten der optimierten Benchmarkprogramme

8.2.3 Aufwand der Registerallokation

Anschließend soll der Zusatzaufwand der Registerallokation während der Codegenerierung festgehalten werden. Das TL-Standardmodul *iter.tm* wurde mit zwei verschiedenen Übersetzern für die Registermaschine übersetzt, einer mit Registerallokation und einer ohne Registerallokation (Operanden werden zwar kurz in Register geladen, aber eigentlich im Keller gehalten). Hier wurden folgende Zeiten für die Codegenerierung gemessen:

- Codegenerierung ohne Registerallokation: 10.09 s
- Codegenerierung mit Registerallokation: 41.89 s

Diese Zeiten entsprechen dem Aufwand der drei zusätzlichen Läufe durch den TML-Baum (Berechnung der Ordnungszahlen, Lebendigkeitsanalyse und Registerzuordnung).

8.2.4 Fazit

Der Code der virtuellen Registermaschine ist nicht bedeutend schneller und oft langsamer als der Code der virtuellen Kellermaschine. Der Hauptgrund hierfür ist, daß die simulierten Register keinen Geschwindigkeitsvorteil über einem simulierten Keller bieten. Es zeigt sich, daß auch Code mit recht guten Registerallokationsergebnissen kaum schneller als Kellermaschinencode ist. Zusätzlich ist der Aufwand für die Registerallokation erheblich höher und erfordert folgende Phasen, um annähernd die gleiche Leistung wie die des simulierten Kellermaschinencodes zu erreichen:

1. Konversion der Abstraktionen in explizite Schritte zum Anlegen von Funktionsobjekten als weitere TML-Transformation
2. Optimierung der TML-Repräsentation
3. Lebendigkeitsanalyse
4. Registerzuordnungslauf
5. Codeerzeugung

Die Codeerzeugung für die Kellermaschine ist einfacher, da nur ein Lauf durch den Baum nötig ist, und die funktionale Natur von TML gut zum Kellerprinzip paßt. Einen Vorteil kann die Registermaschine dennoch besitzen: Bei einer Übersetzung des virtuellen Maschinencodes in echten Maschinencode können die simulierten Register im Gegensatz zum Kellerspeicher einfach auf reale Maschinenregister abgebildet werden. Ob das zu so deutlich höheren Ausführungsgeschwindigkeiten führt, daß der Aufwand der Registerallokation sich lohnt, muß allerdings vorher untersucht werden.

Gemessen an der geringen Performanzsteigerung ist der Aufwand der Codeerzeugung für die Registermaschine zu groß und fehleranfällig, um den vollen Einsatz der TRM als Ausführungsplattform für TooL zu rechtfertigen.

9. Zusammenfassung und Ausblick

Der dynamische Methodenaufruf, Schlüssel zur Ausdrucksmächtigkeit objektorientierter Sprachen, ist Ursache für die schwache Ausführungsgeschwindigkeit und schwierige Optimierbarkeit von TooL-Programmen. Methodenaufrufe sind langsamer als statisch gebundene Prozeduraufrufe, und Optimierungen sind in der Regel vor der Laufzeit nicht wirkungsvoll oder sogar unmöglich. Rein objektorientierte Sprachen sind erheblich stärker von diesen Einschränkungen betroffen als hybride Sprachen mit imperativen und objektorientierten Elementen, da in ersteren auch Kontrollstrukturen und Variablenzugriffe über Botschaften implementiert sind. TooL ist eine rein objektorientierte Sprache, in der Berechnungen ausschließlich über dynamische Methodenaufrufe erfolgen. Damit wird ein objektorientierter Modellierungsansatz von der Sprache vollständig unterstützt, und der Sprachkern bleibt mit wenigen Grundkonstrukten klein. In dieser Arbeit wurde ein Verfahren vorgestellt, mit dem TooL-Programme optimiert werden können. Die Optimierungsstrategie kann durch folgende Punkte zusammengefaßt werden:

- Eine Klassenanalyse gestützt von Profilinformation und einem begrenzten Satz von markierten Typen wie Int, Bool, Fun, Char und String ersetzt dynamische Methodenaufrufe durch statische Prozeduraufrufe.
- Eine erweiterte Beta-Reduktion integriert anschließend kurze Methoden an statisch gebundenen Aufruforten und überführt Endrekursionen in Iterationen.
- Ein zentraler Methodencache enthält optimierte Methoden und sorgt dafür, daß optimierte Methoden nur für zulässige Empfänger aufgerufen werden.
- Sowohl die Profilinformation als auch der zentrale Methodencache sind persistent. Somit kann Laufzeitinformation über mehrere Sitzungen hinweg festgehalten werden, und nach einer Optimierungsphase liegt ein optimierter Methodencache nach jedem Systemstart vor.

Dieses Optimierungsverfahren erzielt recht gute Performanzsteigerungen um einen Faktor von zwei bis drei, wobei der Codezuwachs nur bei 1,76 liegt. Die Frequenz von Methodenaufrufen wird um den Faktor 1,6 verringert. Hölzle berichtet für den SELF-Optimierer, daß dieser die Frequenz der Methodenaufrufe im Schnitt um den Faktor 3,6 verringert [Hölzle 94]. Das schwächere Ergebnis des TooL-Optimierers kann an einer fehlenden Klassenabschätzung liegen, die bewußt ausgelassen wurde, um eine Vergrößerung des Codes durch Klassentests und einen Codeverschiebungslauf (*splitting*) zu vermeiden.

Als Alternative zur Klassenabschätzung bietet sich aufgrund der Erfahrungen dieser Arbeit folgendes an: Die Typinferenz des TooL-Typprüfers ermittelt mit Hilfe der Typdeklarationen

im Quelltext die Signaturen aller Ausdrücke und Bezeichner. Die Information dieser Signaturen kann damit der TML-Generierungsphase zur Verfügung stehen. In dieser Phase kann für einige Methodenaufrufe eine Expansion erfolgen, wenn die Empfängertypen zu den markierten Typen zählen. Auf diese Weise können schon vor dem profilgesteuerten Optimierungslauf Aufrufe mit den Basisselektoren wie „+,-,*“ usw. ersetzt werden, ohne daß auf eine Klassenabschätzung zurückgegriffen werden muß.

Die Möglichkeit, durch aufwendigere Codegenerierung für eine virtuelle Registerarchitektur bessere Laufzeiten zu erzielen, wurde ebenfalls untersucht. Die virtuelle Kellermaschine TVM wurde durch eine Registermaschine mit 16 Registern und einem Zwei-Adreß-Format ersetzt. Die Optimierungsstrategie der Codegenerierung ist folgendermaßen:

- Ein Registerzuordnungsverfahren ermittelt aus festgelegten Registernutzungen von Variablen geeignete Definitionsregister, um Registerbewegungen zu minimieren.
- Während der Codeemission werden die Vorschläge der Registerzuordnung von dem Codegenerator befolgt, wenn diese keinen Lebendigkeitskonflikt ergeben.

Das Verfahren erzeugt recht guten Code mit wenigen Redundanzen. Allerdings läuft dieser nicht nennenswert schneller als der Kellermaschinencode der TVM. Der Hauptgrund hierfür ist, daß die Register der virtuellen Maschine im Hauptspeicher liegen. Ferner behindert die hohe Aufrufdichte von TooL- oder TL-Programmen die Effektivität der Registerallokation. Weiterhin ist die Codegenerierung für eine Registermaschine erheblich aufwendiger und damit komplexer als für eine Kellermaschine, so daß der Einsatz einer virtuellen Registermaschine für Sprachen wie TooL oder TL nicht geeignet ist.

Aus diesen beiden Ergebnissen läßt sich eine mögliche Nachfolgearchitektur des TooL-Backends gewinnen: Ein TML-Baum, in dem schon zur Übersetzung Botschaften an markierte Typen eliminiert werden, wird in einen Kellermaschinenbytecode übersetzt. Dieser kann entweder interpretiert oder vor der Ausführung ähnlich wie in [Deutsch, Schiffman 84] dynamisch in realen Maschinencode übersetzt werden. Methoden in Maschinencode können in einem Cache gehalten und bei Bedarf überschrieben werden, wenn die Konvertierung von Bytecode in Maschinencode schnell verläuft. Erste Ergebnisse mit Maschinencode, der über eine C-Code-Generierung mit Hilfe eines C-Übersetzers erzeugt wurde, zeigen, daß durch die Elimination der TVM-Interpreterschleife Leistungssteigerungen um einen Faktor von zwei bis sechs möglich sind [Weikard 96]. Durch eine profilgesteuerte Optimierungsphase werden weitere Botschaften und Endrekursionen eliminiert. Der optimierte Bytecode könnte in dieser neuen Architektur vor der Ausführung ebenfalls in Maschinencode konvertiert und in dem Cache gehalten werden.

Das Ergebnis der vorliegenden Arbeit ist, daß der konzeptuelle Vorteil eines rein objektorientierten Sprachentwurfs durch die Technologie des TooL-Optimierers unterstützt wird. Der Anspruch der reinen Objektorientierung wird durch die Optimierungsmechanismen nicht verletzt. Damit dient TooL neben Smalltalk und SELF als weiteres Beispiel für eine praktikable, rein objektorientierte Sprache, die für reale Anwendungen geeignet ist.

A. Optimierungsphasen am Beispiel der while-Methode

Im folgenden sollen die einzelnen Optimierungsphasen anhand der while-Methode noch einmal vorgeführt werden. Der Code ist den Ausgaben des Übersetzers ohne Veränderungen entnommen.

- Der ursprüngliche TML-Code der Methode läßt sich aus einem Eintrag in der Profiltabelle für die Empfängerklasse `Nil` auffinden. Alle Operationen sind durch Methodenaufrufe realisiert.

```
proc(self_2 cond_3 statement_4 c_1)
  (send "[]" cond_3 cont(t_5)
  (send "?" t_5
  | proc(c_7)
  | : (send "[]" statement_4 cont(t_8)
  | : (send "while" self_2 cond_3 statement_4 cont(t_9)
  | : (c_7 t_9)))
  | cont(t_10)
  | : (c_1 t_10)))
```

- Nach einer Klassenanalyse sind einige dynamische Methodenaufrufe durch statisch gebundene Aufrufe ausgetauscht worden. Aufrufe an die Methode `[]` der Klasse `Fun` sind durch die direkten Funktionsaufrufe `(cond_905 ..)` und `(statement_906 ..)` ersetzt worden. Folgende Funktionen sind in das Y-Konstrukt aufgenommen:

  – False:?

  ```
  proc(self_918 ifTrue_919 c_920)
    (c_920 ok)
  ```

  – True:?

  ```
  proc(self_922 ifTrue_923 c_924)
    (ifTrue_923 cont(t_925)
    (c_924 t_925))
  ```

```
proc(c_927)
  (Y proc(c_928 False:?_921 True:?_926 Nil:while_913)
  (c_928
  | cont()
  | : (c_927 Nil:while_913)
  | proc(self_918 ifTrue_919 c_920)
  | : (c_920 ok)
  | proc(self_922 ifTrue_923 c_924)
  | : (ifTrue_923 cont(t_925)
  | : (c_924 t_925))
  | proc(self_904 cond_905 statement_906 c_907)
  | : (cond_905 cont(t_908)
  | : (lambda(a:?:1365_914)
  | : | . (lambda(a:?:1366_915)
  | : | . | : (classTest t_908 "False" "True"
  | : | . | : | cont()
  | : | . | : | . (False:?_921 t_908 a:?:1365_914 cont(t_916)
  | : | . | : | . (a:?:1366_915 t_916))
  | : | . | : | cont()
  | : | . | : | . (True:?_926 t_908 a:?:1365_914 cont(t_917)
  | : | . | : | . (a:?:1366_915 t_917)))
  | : | . | cont(t_912)
  | : | . | : (c_907 t_912))
  | : | proc(c_909)
  | : | . (statement_906 cont(t_910)
  | : | . (Nil:while_913 self_904 cond_905 statement_906 cont(t_911)
  | : | . (c_909 t_911)))))))
```

- Im Resultat der Beta-Reduktionsphase ist erkennbar, daß der endrekursive Aufruf an `Nil:while_913` in einen rekursiven Fortsetzungsaufruf `recurse_946` überführt ist:

```
proc(c_927)
  (Y proc(c_928 Nil:while_913)
  (c_928
  | cont()
  | : (c_927 Nil:while_913)
  | proc(PARAM:self_943 PARAM:cond_944 PARAM:statement_945 c_907)
  | : (Y proc(c_947 recurse_946)
  | : (c_947 recurse_946 cont()
  | : (PARAM:cond_944 cont(t_908)
  | : (== t_908 false true
  | : | cont()
  | : | . (c_907 ok)
  | : | cont()
  | : | . (PARAM:statement_945 cont(t_936)
  | : | . (recurse_946)))))))
```

- Zum Vergleich sei der nahezu äquivalente TML-AST eines Aufrufs des TL-while-Schlüsselwortes herangezogen. Es ist deutlich erkennbar, daß das innere Y-Konstrukt obiger TooL-Schleife mit dem des TL-while-Baums übereinstimmt:

```
proc(cond_6 statement_7 e_4 c_5)
  (lambda(whileEnd_9)
  | : (Y proc(c_12 while_8)
  | : (c_12
  | : | cont()
  | : | . (while_8)
  | : | cont()
  | : | . (cond_6 e_4 cont(t_11)
  | : | . (== t_11 true
  | : | . | cont()
  | : | . | : (statement_7 e_4 cont(t_10)
  | : | . | : (while_8))
  | : | . | whileEnd_9))))
  | cont()
  | : (c_5 ok)))
```

- Da der TRM-Codegenerator nicht in dem TooL-Übersetzer eingebaut ist, soll der folgende TRM-Code der TL-Schleifenfunktion illustrieren, daß rekursive Fortsetzungen in Sprünge übersetzt werden:

```
0               0       push            r0              ;
1               1       push            r1              ; cond
2               2       push            r2              ; statement
241  0          3       branch          while8;
                while8:
80  1           5       ld_sp           r0       1      ; cond
48              7       call             0              ; cond
211  240        8       ld_true         r15             ;
139  15         10      eq              r0      r15     ;
242  2          12      branch_cf               l0;
241  5          14      branch          whileEnd9;
                l0:
80  0           16      ld_sp           r0       0      ; statement
48              18      call             0              ; statement
241  ~16        19      branch          while8;
                whileEnd9:
209  0          21      ld_ok           r0              ;
67              23      return           3              ;
```

B. TML-Primitive

B.1 Ursprüngliche TML-Primitive im Tycoon-Backend

(Y λ($c\ v_1 \ldots v_n$) app)	Y-Kombinator für rekursive Definitionen
($p\ x\ y\ c_e\ c_c$)	Ganzzahlige Arithmetik, $p \in \{+, -, *, /, \%\}$
($p\ x\ y\ c_1\ c_2$)	Vergleichsoperationen, $p \in \{<, >, <=, >=\}$
($p\ x\ y\ c$)	Bit-Operationen, $p \in \{<<, >>, \&, \mid, \hat{}, \tilde{}\}$
(== x $v_1 \ldots v_n$ $c_1 \ldots c_n\ [c_{n+1}]$)	Konditional, beruhend auf Objektidentität, mit Werten und Verzweigungen (optionaler *else*-Zweig)
(char2int $x\ c$)	Konversion eines Zeichens in einen ganzzahligen Wert
(int2char $x\ c$)	Konversion eines ganzzahligen Werts in ein Zeichen
(array $val_1 \ldots val_n\ c$)	Anlegen eines veränderlichen Feldes mit n Einträgen
(vector $val_1 \ldots val_n\ c$)	Anlegen eines unveränderlichen Feldes mit n Einträgen
(new n *init* c)	Anlegen eines veränderlichen Feldes mit n initialisierten Einträgen
($new n *init* c)	Anlegen eines veränderlichen Bytefeldes mit n initialisierten Einträgen
([ ] $x\ i\ c$)	Feldzugriff
([ ] := $x\ i\ v\ c$)	Feldzuweisung
($[ ] $x\ i\ c$)	Bytefeldzugriff
($[ ] := $x\ i\ v\ c$)	Bytefeldzuweisung
(size $v\ c$)	Größe eines Feldes
(move n *src srcOffset dst dstOffset* c)	Verschieben von Feldern
($move n *src srcOffset dst dstOffset* c)	Verschieben von Bytefeldern
(ccall *fmt cfn* $c_1\ c_2$)	C-Funktionsaufruf
(pushHandler $c_1\ c$)	Festlegung der Fortsetzung c_1 als Ausnahmebehandlung
(popHandler c)	Entfernen der aktuellen Ausnahmebehandlung
(raise x)	Melden des Ausnahmezustandes x

B.2 TooL-spezifische TML-Primitive

(send *sel r* $arg_1 \ldots arg_n$ *c*)	Aufruf der Botschaft *sel* mit Empfänger *r*
(sendSuper *sel self* $arg_1 \ldots arg_n$ *c*)	Aufruf der Botschaft *sel* an eine Superklasse
(classTest *v*	Konditional, beruhend auf
$class_1 \ldots class_n$	Klassenzugehörigkeit von *v*
$c_1 \ldots c_n$ [c_{n+1}])	mit Verzweigungen (optionaler *else*-Zweig)
(classnew *class n c*)	Anlegen eines Exemplars von *class* mit *n* slots
($classnew *class n c*)	Anlegen eines Exemplars einer Byteklasse

C. TRM-Befehlssatz

Opcode (Hex)	*Befehl*	*Bedeutung*
0i	`push`	Register i im Keller ablegen
1i	`drop`	i Kellerfelder entfernen
2i	`jump`	Endrekursiver Sprung an das Codeobjekt in r0
3i	`call`	Aufruf des Codeobjekts in r0
4i	`return`	Return, i Kellerfelder entfernen
5d ii	`ld_sp`	rd $\leftarrow$ Kellerfeld ii
6d ii	`ld_lit`	rd $\leftarrow$ Literal ii
7i ds	`seti`	rd[i] $\leftarrow$ rs
80 ds	`get`	rd $\leftarrow$ rd[rs]
8x ds	`alu`	rd $\leftarrow$ rd $\otimes$ rs,
		$\otimes \in \{+, -, \times, \div, mod, \wedge, \vee, xor, shl, shr, =, >, \geq, <, \leq\}$
90 ds	`geti`	rd $\leftarrow$ rd[i]
9x di	`alui`	rd $\leftarrow$ rd $\otimes$ i,
		$\otimes \in \{+, -, \times, \div, mod, \wedge, \vee, xor, shl, shr, =, >, \geq, <, \leq\}$
ad ii	`closure`	rd $\leftarrow$ neues Feld der Größe ii
bi jj	`send`	send r0 Selektor: jj, Argumente in r1...r15
ci ii jj	`sendLong`	send r0 Selektor: iijj, Argumente in r1...r15
d0 d-	`set_immutable`	TSP-Objekt in rd als unveränderlich kennzeichnen
d1 d-	`ld_ok`	rd $\leftarrow$ ok (TSP-Wert für nil)
d2 d-	`ld_false`	rd $\leftarrow$ false
d3 d-	`ld_true`	rd $\leftarrow$ true
d4 di ii	`ld_sp2`	rd $\leftarrow$ Kellerfeld iii
d5 di ii	`ld_lit2`	rd $\leftarrow$ Literal iii
d6 ds ii	`seti2`	rd[ii] $\leftarrow$ rs
e0 di	`ld_int`	rd $\leftarrow$ i
e1 ds	`mov`	rd $\leftarrow$ rs
e2 ds	`xchg`	rd $\leftrightarrow$ rs
e3 ds	`size`	rd $\leftarrow$ TSP_size(rs)
e4 ds	`not`	rd $\leftarrow \neg$rs
e5 ds	`neg`	rd $\leftarrow$ -rs
e6 ds	`char2int`	rd $\leftarrow$ Integer(rs)
e7 ds	`int2char`	rd $\leftarrow$ Char(rs)
e8 ds	`get_byte`	rd $\leftarrow$ Bytezugriff: rd[rs]

Opcode (Hex)	*Befehl*	*Bedeutung*
e9 ds	`array`	rd ← neues TSP-Feld der Größe rs initialisiert mit rd
ea ds	`byte_array`	rd ← neues TSP-Bytefeld der Größe rs initialisiert mit rd
eb	`push_handler`	Ausnahmebehandlung im Ausnahmekeller speichern
ec	`pop_handler`	Ausnahmebehandlung löschen
ed	`set`	r0[r1] ← r2
ee	`set_byte`	Bytezugriff: r0[r1] ← r2
ef	`block_move`	r0 Worte vom Feld in r1 beim Index r2 ins Feld r3 ab dem Index r4 verschieben
f0	`block_move_bytes`	r0 Bytes vom Feld in r1 beim Index r2 ins Feld r3 ab dem Index r4 verschieben
f1 ii	`branch`	Sprung um ii Bytes
f2 ii	`branch_cf`	Bedingter Sprung um ii Bytes
f3 iijj	`long_branch`	Sprung um iijj Bytes
f4 iijj	`long_branch`	Bedingter Sprung um iijj Bytes
f5	`raise`	Ausnahme auslösen
f6	`ccall`	C-Funktionsaufruf mit *Bibliothek:Funktionsname*, *Übergabeformatstring* und Argumenten im Keller
fb ds	`class_new`	rd ← neues Exemplar der Klasse rd mit rs Slots
fc ds	`byte_class_new`	rd ← neues Exemplar der Byteklasse rd mit rs Slots
fd iijj	`sendSuper`	send Super(r0) Selektor: iijj, Argumente in r1...r15

Literaturverzeichnis

Abadi, Cardelli 95: Abadi, M. und Cardelli, L. „On Subtyping and Matching". In: *ECOOP '95 Conference Proceedings.* Springer-Verlag, Oktober 1995.

Aho et al. 86: Aho, A., Sethi, R., und Ullman, J.D. *Compilers: Principles, Techniques, Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

Appel 89: Appel, A. W. „Continuation-Passing, Closure-Passing Style". In: *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Januar 1989, S. 293–302.

Atkinson 86: Atkinson, R. G. „Hurricane: An Optimizing Compiler for Smalltalk". In: *OOPSLA '86 Conference Proceedings*, Portland,OR., September 1986, S. 151–158. Published as SIGPLAN Notices 21(11).

Auslander, Hopkins 82: Auslander, M und Hopkins, M. „An Overview of the PL.8 Compiler". In: *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, Boston, MA., Juni 1982.

Bartley, Jensen 86: Bartley, D.H. und Jensen, J.C. „The Implementation of PC Scheme". In: *Proceedings of the ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, 1986, S. 86–93.

Bobrow et al. 88: Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., und Moon, D.A. „Common Lisp Object System Specification". Technical Report ANSI Document X3J13 88-002R, Juni 1988. In: SIGPLAN Notices 23 (Special Issue), September 1988.

Bruce 96: Bruce, K. „Typing in object-oriented languages: Achieving expressiveness and safety". Technical report, Williams College, 1996. erscheint in: Computing Surveys. ftp: cs.williams.edu/pub/kim.

Cardelli et al. 89: Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., und Nelson, G. „Modula-3 Report (revised)". Report 52, DEC SRC, Palo Alto, November 1989.

Chaitin et al. 81: Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., und Markstein, P.W. „Register allocation via coloring". *Computer Languages*, Jg. 6, 1981, S. 47–57.

Chambers, Ungar 89: Chambers, C. und Ungar, D. „Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language". In: *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR., Juli 1989.

Chambers 92: Chambers, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Dissertation, Stanford University, 1992.

Chambers 93: Chambers, C. „The Cecil language – Specification and Rationale". Technical Report TR 93-03-05, Dept. of Computer Science and Engineering, University of Washington, März 1993.

Chow, Hennessy 90: Chow, F.C. und Hennessy, J.L. „The Priority-Based Coloring Approach to Register Allocation". In: *ACM Trans. on Programming Languages and Systems*, Bd. 12:4, Oktober 1990.

Connor et al. 90: Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A., und Morrison, R. „The Persistent Abstract Machine". In: *Persistent Object Systems.* Springer-Verlag, 1990.

Cox 86: Cox, B. J. *Object Oriented Programming - An Evolutionary Approach.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

Cutts et al. 94: Cutts, Q. I., Connor, R.C.H., Kirby, G.N.C., und Morrison, R. „An Execution-Driven Approach to Code Optimisation". In: *Proceedings of the 17th Australasian Computer Science Conference*, Christchurch, New Zealand, 1994, S. 83–92.

Deutsch, Schiffman 84: Deutsch, L.P. und Schiffman, A. „Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT., 1984.

Driesen 93: Driesen, K. „Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages". Diplomarbeit, Vrije Universiteit Brussel, 1993.

Ellis, Stroustrup 90: Ellis, M.A. und Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

Engler, Proebsting 94: Engler, D. R. und Proebsting, T.A. „DCG: An Efficient, Retargetable Dynamic Code Generation System". In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oktober 1994, S. 263–273.

Gawecki, Matthes 94: Gawecki, A. und Matthes, F. „The Tycoon Machine Language TML - An Optimizable Persistent Program Representation". FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Juli 1994.

Gawecki, Matthes 96: Gawecki, A. und Matthes, F. „Integrating Subtyping, Matching and Type Quantification: A Practical Perspective". In: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP '96*, Linz,Austria, Juli 1996.

Gawecki 92: Gawecki, A. „Ein optimierender Übersetzer für Smalltalk". Bericht FBI-HH-B-152/92, Fachbereich Informatik, Universität Hamburg, September 1992.

Goldberg, Robson 83: Goldberg, A. und Robson, D. *Smalltalk–80: The Language and its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

Gosling, McGilton 95: Gosling, J. und McGilton, H. „The Java Language Environment White Paper". Technical report, Sun Microsystems, Inc., 1995.

Grove et al. 95: Grove, D., Dean, J., Garrett, C., und Chambers, C. „Profile-Guided Receiver Class Prediction". In: *OOPSLA '95 Conference Proceedings*, Oktober 1995.

Hennessy, Patterson 90: Hennessy, J.L. und Patterson, D.A. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, California, 1990.

Hölzle 94: Hölzle, U. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Dissertation, Stanford University, 1994.

Hutchinson 87: Hutchinson, N. *Emerald: An Object-Based Language for Distributed Programming.* Dissertation, Department of Computer Science, University of Washington, Seattle, Januar 1987. Technical Report: 87-01-01.

Ingalls 78: Ingalls, D.H.H. „The Smalltalk-76 Programming System: Design and Implementation". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ., Januar 1978, S. 9–16.

Johnson et al. 88: Johnson, R.E., Graver, J.O., und Zurawski, L.W. „TS: An Optimizing Compiler for Smalltalk". In: *OOPSLA '88 Conference Proceedings*, San Diego, CA., Oktober 1988, S. 18–26. Published as SIGPLAN Notices 23(11).

Kelsey, Hudak 89: Kelsey, R.A. und Hudak, P. „Realistic Compilation By Program Transformation". In: *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Januar 1989, S. 281–292.

Kiradjiev 94: Kiradjiev, P. „Dynamische Optimierung in CPS-orientierten Zwischensprachen". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Dezember 1994.

Kornacker 95: Kornacker, M. „Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, August 1995.

Kranz et al. 86: Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., und Adams, N. „ORBIT: An Optimizing Compiler for Scheme". *ACM SIGPLAN Notices*, Jg. 21, Juli 1986, Nr. 7, S. 219–233.

Krasner 83: Krasner, G. *Smalltalk-80 Bits of History, Words of Advice.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

Lee, Leone 96: Lee, P. und Leone, M. „Optimizing ML with Run-Time Code Generation". In: *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Mai 1996.

Levesque, Brachman 85: Levesque, H.J. und Brachman, R.J. „A Fundamental Tradeoff in Knowledge Representation and Reasoning". In: *Readings in Knowledge Representation.* Morgan Kaufmann Publishers Inc., San Mateo, California, 1985, S. 42–70.

Mathiske et al. 95: Mathiske, B., Matthes, F., und Schmidt, W., J. „On Migrating Threads". In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, Juni 1995. Also appeared as TR FIDE/95/136, FIDE Technical Report Series.

Matthes, Schmidt 92: Matthes, F. und Schmidt, J.W. „Definition of the Tycoon Language TL – A Preliminary Report". Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, November 1992.

Matthes, Schmidt 94: Matthes, F. und Schmidt, J. W. „Persistent Threads". In: *Proceedings of the 20th Conference on Very Large Databases, VLDB '94*, Santiago, Chile, September 1994.

Meyer 88: Meyer, B. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Morrison et al. 94: Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., und Munro, D.S. „The Napier88 Reference Manual (Release 2.0)". FIDE Technical Report FIDE/94/104, University of St. Andrews, 1994.

Reade 89: Reade, C. *Elements of Functional Programming.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

Römer, Lotter 93: Römer, T. und Lotter, B. „Ein generisches Wörterbuch beliebiger Dimension in Tycoon". Studienarbeit, Fachbereich Informatik, Universität Hamburg, November 1993.

Römer, Lotter 96: Römer, T. und Lotter, B. „Prinzipien des Entwurfes von Software-Bibliotheken für Massendatentypen und ihre Umsetzung in das Tycoon-System". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, April 1996.

Steele 78: Steele, G.L. „Rabbit: A Compiler for SCHEME". Technical report, Massachusetts Institute of Technology, Mai 1978.

Stoy 77: Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

Teodosiu 91: Teodosiu, D. „HARE: An Optimizing Portable Compiler for Scheme". *ACM SIGPLAN Notices*, Jg. 26, Januar 1991, Nr. 1, S. 109–120.

Ungar et al. 84: Ungar, D., Blau, R., Foley, P., und Patterson, D. „Architecture of SOAR: Smalltalk on a RISC". In: *Eleventh Annual International Symposium on Computer Architecture*, Ann Arbor, MI., 1984.

Ungar, Smith 87: Ungar, D. und Smith, R.B. „Self: The Power of Simplicity". In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida*, 1987, S. 227 – 242.

Ungar 87: Ungar, D. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, Cambridge, MA, 1987.

Weikard 96: Weikard, M. „Dynamische Maschinencodegenerierung in Tycoon". Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996. in Vorbereitung.

Wulf et al. 73: Wulf, W.A., Johnsson, R.K., Weinstock, C.B., und Hobbs, S.O. „The Design of an Optimizing Compiler". Technical Report AFOSR-TR-74-0096, Carnegie Mellon University, Air Force Office of Scientific Research, 1973.

Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, 27. Juni 1996

Martin Pakendorf