Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz

Ralf Scheidhauer

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

Saarbrücken
Dezember 1998

Ralf Scheidhauer
Forschungsbereich Programmiersysteme
DFKI GmbH
Stuhlsatzenhausweg 3
66123 Saarbrücken
Ralf.Scheidhauer@dfki.de

Dekan:	Prof. Dr. Wolfgang Paul
Vorsitzender:	Prof. Dr. Jörg Siekmann
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Prof. Dr. Reinhard Wilhelm
Tag des Kolloquiums:	18. Mai 1999

Zusammenfassung

Diese Arbeit beschreibt *Design*, *Implementierung* und *Evaluierung* einer virtuellen Maschine für die Kernsprache von Oz, die wir mit L bezeichnen. Wir stellen L aus didaktischen Gründen als Erweiterung einer Teilsprache von SML dar. Die wichtigsten Unterschiede von L zu SML sind: logische Variablen, Threads, Synchronisation und dynamische Typisierung. Ausgehend von einer informellen Beschreibung der dynamischen Semantik über ein Graphenmodell entwickeln wir daraus schrittweise auf unterschiedlichen Abstraktionsebenen eine virtuelle Maschine für L. Wir beschreiben zunächst ein einfaches Grundmodell. Darauf aufbauend diskutieren wir verschiedene Optimierungen. Schließlich verfeinern wir weiter, indem wir auf Aspekte der Implementierung des Modells eingehen. Abschließend evaluieren wir die Effektivität der vorgestellten Techniken an einer Reihe von größeren Anwendungen aus der Praxis. Weiter zeigen wir, daß die Implementierung der Sprache kompetitiv ist mit den schnellsten Emulatoren für statisch getypte funktionale Sprachen.

Abstract

This thesis presents the *design*, *implementation* and *evaluation* of a virtual machine for the core language of Oz, which we call L. We present L for didactic reasons as an extension of a sublanguage of SML. The most important differences between L and SML are: logic variables, threads, synchronization and dynamic typing. Starting from an informal description of the dynamic semantics in terms of a graph model, we develop step by step on various levels of abstraction a virtual machine for L. We begin with a simple basic model. We then propose several optimizations of this model. Afterwards we keep refining our approach by addressing specific aspects of the implementation of the model. Finally we evaluate the effectiveness of the techniques using a set of larger real world applications. Further we show, that the implementation of the language is competitive with the fastest emulators for statically typed functional languages.

Ausführliche Zusammenfassung

Der Autor war zusammen mit Michael Mehl für den Kern der Implementierung der Programmiersprache Oz verantwortlich. Seit 1991 wird Oz am DFKI entwickelt, das Strukturen aus logischer, funktionaler und objektorientierter Programmierung in sich vereint. Entsprechend der Weiterentwicklung von Oz wurden drei Systeme fertiggestellt [Oz95, Oz97, Moz98]. Diese Arbeit befaßt sich mit wesentlichen Ergebnissen, die bei Design, Implementierung und Evaluierung einer virtuellen Maschine für diese Systeme gewonnen wurden. Wir konzentrieren uns auf Teilaspekte der Implementierung von Oz, indem wir auf eine idealisierte Teilsprache von Oz, die wir mit „L“ bezeichnen, zurückgreifen. Bei Gesichtspunkten von Oz, die in L nicht berücksichtigt wurden, handelt es sich um orthogonale Erweiterungen von L; sie werden in eigenen Arbeiten ausführlich beleuchtet.

Wir wollen den Leser nicht unnötig mit dem Erlernen von Syntax und Semantik einer neuen Sprache belasten. Wir führen L daher so ein, indem wir L als Erweiterung einer Teilsprache von SML definieren. Dadurch kann sich der Leser auf die Unterschiede von L gegenüber SML konzentrieren: logische Variablen, Threads, Synchronisation und dynamische Typisierung.

Die dynamische Semantik von L erklären wir informell am Modell eines Graphen. Dieses Modell setzt auf einem niedrigeren Abstraktionsniveau an als die Sprachdefinitionen von Oz, die die Sprache über das Constraintmodell definieren. Das Graphenmodell bietet zwei wesentliche Vorteile: zum einen stellt es einen klaren Ausgangspunkt dar, auf dessen Basis sich eine effiziente Implementierung der Sprache realisieren läßt und gibt daher zum anderen auch ein gutes Kostenmodell.

Aus dem Graphenmodell entwickeln wir dann das Modell einer virtuellen Maschine für L. Unsere virtuelle Maschine ist sequentiell; das heißt, daß die Implementierung der Maschine mit einem Thread die multiplen Threads von L realisiert. Wir gehen schrittweise in unterschiedlichen Abstraktionsstufen vor. Wir beschreiben zunächst ein einfaches Grundmodell, das die wichtigsten Ideen zur Realisierung von logischen Variablen, Threads, Suspension und Synchronisation wiedergibt. Insbesondere stellen wir ein flexibles Format zur Darstellung von Maschinenprogrammen vor, das auch den Erfordernissen für Persistenz und Verteilung gerecht wird.

Aufbauend auf dem noch einfachen Grundmodell gehen wir auf wichtige Optimierungen dieses Modells ein und beschreiben Techniken wie zum Beispiel zur Code-Spezialisierung und -Instantiierung, zur Optimierung der Unifikation, zur effizienten Behandlung von Konstruktoren und Funktionen. Wir stellen insbesondere eine effiziente Implementierung des Synchronisationskonstruktes von L vor, dessen Semantik über logische Implikation definiert ist. Existierende Implementierungen für vergleichbare Konstrukte aus dem Bereich der nebenläufigen Constraint-Programmierung waren entweder ineffizient oder schränkten deren Form und Verwendung ad hoc ein.

In einem weiteren Verfeinerungsschritt gehen wir dann auf wichtige Aspekte der Implementierung ein. Wir beschreiben Techniken zur effizienten Implementierung von Emulatoren, zur Speicherverwaltung, zur Darstellung von Datenstrukturen, zu effizienten Threads und zu nativen Funktionen.

Abschließend evaluieren wir die Effektivität der vorgestellten Techniken. Wir vergleichen Mozart, die Implementierung von Oz, mit den Implementierungen anderer Sprachen. Unter den Emulatoren nimmt Mozart eine Spitzenstellung ein: es ist kompetitiv mit den schnellsten Emulatoren für statisch getypte Sprachen. Mozart ist in der Regel etwa doppelt so schnell wie Sicstus Prolog und Erlang, Sprachen, die von ihrem Charakter her (dynamische Typisierung, Nebenläufigkeit, logische Variablen) eher dem von Oz ähneln als etwa die ML Dialekte. Im Vergleich zu nativen Systemen liegt Mozart zwar bei arithmetischen Benchmarks um bis zu einer Größenordnung zurück. Bei den eigentlichen Einsatzgebieten von Oz wie symbolischen Berechnungen schmilzt dieser Abstand auf einen Faktor von selten mehr als zwei oder drei. In bestimmten Fällen kann Mozart hier sogar schneller sein. Insbesondere zeigt sich, daß die Implementierung von Threads in Mozart besonders effizient und leichtgewichtig ist.

Schließlich untersuchen wir anhand von größeren realen Applikationen das dynamische Verhalten von typischen Oz-Programmen aus unterschiedlichen Anwendungsbereichen und erstellen Profile beispielsweise zur Struktur des Haldenspeichers, der Verwendung von Funktionen, Unifikation und Threads. Diese Zahlen belegen die Effektivität der in dieser Arbeit vorgestellten Techniken in der Praxis, zeigen aber auch Stellen auf, an denen noch Bedarf für Verbesserungen besteht.

Extended Abstract

The author was leading together with Michael Mehl the implementation of the core language of Oz. Since 1991 Oz is being developed at DFKI and integrates features from logic, functional and object-oriented programming. Corresponding to the development phases of Oz three systems have been released [Oz95, Oz97, Moz98]. This thesis presents obtained during the design, implementation and evaluation of a virtual machine for these systems. We concentrate on certain aspects of Oz by using an idealized sublanguage of Oz, which we call "L". Aspects of Oz, that are not covered by L, are extensions orthogonal to L; they are discussed in detail in separate publications.

We do not wish to burden the reader with the necessity to learn yet another programming language. Therefore we present L as an extension of a sublanguage of SML, so that the reader may concentrate on the differences between L and SML, namely logic variables, threads, synchronization and dynamic typing.

We explain the dynamic semantics of L informally in terms of a graph model. The graph model rests on a lower level of abstraction than the constraint model, the language definition of Oz. The graph model offers two substantial advantages: it not only gives a clear starting point for the efficient implementation of the language but also provides a good cost model.

Starting from the graph model we develop incrementally a model of a virtual machine for L. Our virtual machine is sequential, which means that the implementation of the machine implements in a single thread the multiple threads of L. We describe the machine, by means of decreasing levels of abstraction. We first describe a simple basic model showing the key ideas of the implementation of threads, logic variables. suspension and synchronization. In particular we present a flexible machine code format, which also fullfils the requirements for persistence and distribution.

Building on the simple basic model we then introduce important optimizations of this model. We describe techniques for code specialization and instantiation, for the optimization of unification, for the efficient handling of constructors and functions. In particular we present an efficient implementation of the synchronization construct of L, who's semantics is defined via logical entailment. Existing implementations of comparable constructs from the area of concurrent constraint programming languages were either inefficient or imposed ad hoc limitations on their structure and use.

In a further refinement, we then deal with important aspects of the implementation of the machine model. We describe techniques for the efficient implementation of emulators, the memory management, the representation of data structures, particularly light-weight threads and native functions.

Finally we evaluate the effectiveness of the techniques presented. We compare Mozart, the implementation of Oz, with implementations of other languages. Under the emulators Mozart takes a leading place: it is competitive with the fastest emulators for statically typed languages. Mozart is on average about twice as fast as Sicstus Prolog and Erlang, both programming languages whose character (dynamic typing, concurrency, logic variables) is closer to Oz than for instance the ML dialects. Compared with native code systems Mozart is up to one order of magnitude slower for

arithmetic benchmarks; for the primary application areas like symbolic computations this distance melts down to a factor rarely more than two or three; in certain cases Mozart can be even faster. Above all the implementation of threads in Mozart turns out be very efficient and light weight.

Finally we examine the dynamic behavior of typical Oz programs using real life applications and create profiles for the structure of the memory usage, the use of functions, unification and threads for example. These numbers demonstrate the effectiveness of the techniques presented in this thesis in practice, however they also show that in certain places there is still the need for improvements.

Danksagung

Die Ergebnisse, die in dieser Dissertation vorgestellt werden, sind nicht die Resultate der Arbeit eines Einzelnen. Viele Ideen und Lösungen entstanden in zahlreichen Diskussionen mit meinen Kollegen und sind erst in enger Teamarbeit zu dem geworden, wie sie sich aus heutiger Sicht darstellen. Ich möchte mich daher bei allen bedanken, die zum Entstehen dieser Arbeit beigetragen haben. Zu allererst bei meinem Doktorvater Gert Smolka, für seinen Enthusiasmus, seine Unterstützung in allen Belangen und die Schaffung einer außergewöhnlichen Arbeitsumgebung. Sein Streben, komplexe Zusammenhänge zu vereinfachen, war ein steter Ansporn und hat meine ganze Arbeit entscheidend beeinflußt. Bei Michael Mehl für unzählige Diskussionen, in denen er oft die richtigen Ideen hatte, wenn ich nicht mehr weiter kam; Michael kennt die Maschine wie kein Zweiter und hat sich mit seinen oft unorthodoxen Einfällen zum Glück meist gegen meinen eigenen Konservatismus durchgesetzt. Ich danke allen Kollegen des Forschungsbereichs Programmiersysteme, dabei Denys Duchier und Christian Schulte für ihre Unterstützung in Fragen zu Oz wie auch zu Punkten der Implementierung; Martin Henz für die Zusammenarbeit bei der Implementierung des ersten Compilers und der Optimierung des Objektsystems; Leif Kornstaedt für Diskussionen zu vielen Themen der Compilation und Implementierung der Maschine; Tobias Müller, Joachim Walser und Jörg Würtz in Fragen zur Constraint-Programmierung; Martin Müller und Joachim Niehren für ihren Beistand in theoretischen Fragen; Kostja Popov für seine Unterstützung bei der Implementierung, insbesondere des allerersten Prototypen der Maschine; Andreas Rossberg für seine Hilfe zu Fragen, die SML betrafen. Denys Duchier, Andreas Rossberg und Christian Schulte danke ich insbesondere für Kommentare zu früheren Fassungen der Arbeit. Per Brand, Seif Haridi und Peter Van Roy danke ich dafür, daß sie mit mir ihre Erfahrungen vor allem auch in Fragen zur Implementierungstechnik teilten.

Ralf Scheidhauer
Saarbrücken
Dezember 1998

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Logische Variablen 2
1.2 Threads 2
1.3 Synchronisation 3
1.4 Schnelle Emulatoren 4
1.5 Beiträge 5
1.6 Gliederung der Arbeit 7

I Die Sprache **9**

2 Die Sprache L **11**

2.1 Überblick über die Sprache L 12
2.2 Notation 13
2.3 Die Sprachen L– und L 14
2.4 Der Speicher als Graph 15
2.5 Werte und Knoten 15
2.6 Gleichheit 18
2.7 Umgebungen und Abschlüsse 18
2.8 Auswertung von Ausdrücken 19
2.9 Operationen in L– 23
2.9.1 Arithmetik 23
2.9.2 Zellen 23
2.10 Logische Variablen 25
2.10.1 Erzeugung 25
2.10.2 Unifikation 25
2.10.3 Zyklische Datenstrukturen 25

2.10.4 Suspension 27
2.10.5 Case 28
2.11 Threads 30
2.11.1 Fairneß 31
2.11.2 Monotonie 31
2.11.3 Erzeugung 32
2.12 Fehlerbehandlung 32
2.13 Beispiele 32
2.13.1 Top-down Konstruktion von funktionalen Datenstrukturen 32
2.13.2 Sichere abstrakte Datentypen 33
2.13.3 Synchronisierte Schlange 34

II Maschinenmodell 39

3 Grundmodell 41
3.1 Eine neue Sprache 42
3.2 Zusammenspiel Übersetzer und Maschine 43
3.3 Der Graph im Maschinenmodell 44
3.4 Umgebungen und Register 46
3.5 Maschinenprogramme 47
3.5.1 Prelude 48
3.5.2 Segmente 49
3.5.3 Beispiel 49
3.5.4 Der Lader 50
3.5.5 Threads und Aufträge 51
3.6 Signalbehandlung 52
3.7 Suspension 53
3.7.1 Suspensionslisten und Suspensionen 54
3.7.2 Suspension auf Gleichheit zweier Variablen 56
3.8 Instruktionen 56
3.8.1 Operatoren 58
3.8.2 Deklaration 59
3.8.3 Funktionen 59
3.9 Case 65

3.9.1 Was muß die Implementierung leisten? . . . 65
3.9.2 Prinzipielles Vorgehen . . . 67
3.9.3 Instruktionen . . . 68
3.9.4 Beispiel . . . 72
3.10 Inkrementalität und getrennte Übersetzung . . . 73
3.10.1 Ausführung mehrerer Programme . . . 73
3.10.2 Export von Variablen . . . 74
3.10.3 Die Exporttabelle . . . 75
3.10.4 Optimierungen über Programmgrenzen . . . 76
3.11 Persistenz . . . 77
3.12 Verteilung . . . 78

4 Optimierungen **81**
4.1 Codespezialisierung . . . 81
4.1.1 Statische Spezialisierung . . . 82
4.1.2 Dynamische Spezialisierung . . . 83
4.1.3 Zeitaufwand . . . 84
4.1.4 Unterschiedliche Instruktionslängen . . . 84
4.1.5 Ersetzung von Instruktionsfolgen . . . 85
4.1.6 Kopieren von Code . . . 85
4.2 Unifikation . . . 88
4.3 Tupelkonstruktion . . . 88
4.3.1 Argumentzeiger . . . 89
4.3.2 Modus-Register . . . 90
4.3.3 Lese/Schreib-Unifikation außerhalb von Wächtern . . . 91
4.3.4 Listen . . . 91
4.4 Konstruktoren . . . 92
4.5 Arithmetik . . . 93
4.6 Case . . . 93
4.6.1 Einfache Tests . . . 93
4.6.2 Boolesche Tests . . . 96
4.6.3 Redundante Tests . . . 97
4.6.4 Indexierung . . . 97
4.7 Funktionen . . . 99

4.7.1 Endrekursion 99
4.7.2 Applikation 100
4.7.3 Allokation der L-Register 102
4.7.4 Programmtransformation 104

5 Diskussion **107**
5.1 Stack- oder Registermaschine 107
5.2 Der Keller 108
5.3 Referenzketten 109
5.4 Funktionen 112
5.4.1 Darstellung der Abschlüsse 112
5.4.2 Wegfall der G-Adressierung 113
5.5 Case 114
5.6 Registerallokation 115
5.6.1 Zuordnung der Adressierungsart 115
5.6.2 Temporäre und permanente Variablen 116
5.6.3 L-Register 116
5.6.4 X-Register 118
5.7 Lazy Evaluierung 119

III Implementierung **121**

6 Der Emulator **123**
6.1 Überblick 124
6.2 Maschinencode 125
6.3 Zustände des Emulators 127
6.4 C++ als Implementierungssprache 127
6.5 Der Interpreter 128
6.6 Threaded Code 129
6.7 Native Codeerzeugung 132
6.8 Verschiedene Optimierungs-Techniken für Emulatoren 134
6.8.1 Zusammenfassung von Instruktionen 134
6.8.2 Spezialisierung von Instruktionen 135
6.8.3 Register-Zugriffe 135

6.8.4 Zuordnung virtueller Maschinenregister zu realen . . . 135
6.8.5 Optimierung des C++ Codes . . . 135
6.8.6 Cacheverhalten . . . 137

7 Der Speicher 139
7.1 Speicherverwaltung . . . 139
7.1.1 Die Halde . . . 139
7.1.2 Freispeicherverwaltung . . . 140
7.1.3 Speicherbereinigung . . . 143
7.2 Markierte Referenzen . . . 145
7.2.1 Wahl und Darstellung der Marken . . . 147
7.2.2 Das Tag-Schema von Mozart . . . 147
7.2.3 Wahl der Marken . . . 149
7.2.4 Vergrößerung des Adreßraums . . . 150
7.3 Darstellung der Werte . . . 150
7.3.1 Zahlen . . . 151
7.3.2 Literale . . . 152
7.3.3 Zellen . . . 153
7.3.4 Tupel . . . 154
7.3.5 Funktionen . . . 155
7.4 Variablen . . . 156
7.4.1 Binden von Variablen . . . 156
7.4.2 Binden an Nicht-Variablen . . . 158
7.4.3 Aufsplittung . . . 159
7.4.4 Variablen in Argumenten von Tupeln . . . 159
7.4.5 Kellervariablen . . . 160
7.4.6 Variablen in Mozart . . . 161
7.4.7 Variablen in der WAM . . . 163
7.4.8 Variablen in L . . . 164
7.5 Rationale Unifikation . . . 164
7.6 Lokale Variablen im Case . . . 164

8 Threads 169
8.1 Darstellung im Speicher . . . 169
8.2 Cachen des obersten Auftrages . . . 170

8.3 Fairneß . . . 171
8.4 Signalbehandlung . . . 171
8.5 X-Register . . . 172
8.6 Besondere Aufträge . . . 173
8.7 Threads und Effizienz . . . 174

9 Native Funktionen 177
9.1 Darstellung im Speicher . . . 177
9.2 Applikation . . . 178
9.3 Determiniertheit . . . 179
9.4 Suspension . . . 179
9.5 Operationen und Builtins . . . 179

IV Evaluierung 181

10 Evaluierung 183
10.1 Das Problem eines fairen Vergleichs . . . 184
10.2 Sprachen und Systeme . . . 187
10.3 Die Testplattform . . . 189
10.4 So haben wir gemessen . . . 189
10.5 Quellprogramme . . . 190
10.6 Mikro Benchmarks . . . 190
10.7 Standard-Benchmarks . . . 191
10.7.1 Takeushi . . . 191
10.7.2 Fibonacci . . . 193
10.7.3 Naive Reverse . . . 195
10.7.4 Quicksort . . . 196
10.7.5 N-Damen . . . 198
10.7.6 Mandelbrot . . . 199
10.7.7 Symbolische Ableitung . . . 200
10.7.8 Threads . . . 201
10.8 Große Anwendungen . . . 203
10.8.1 Haldenspeicher . . . 205
10.8.2 Erzeugung von Funktionen . . . 206

10.8.3 Funktionsaufrufe . . . 206
10.8.4 Unifikation . . . 208
10.8.5 Threads . . . 209

11 Verwandte Arbeiten 213
11.1 Multilisp und Futures . . . 213
11.2 Concurrent ML . . . 214
11.3 WAM . . . 215
11.4 AKL . . . 215
11.5 Erlang . . . 216
11.6 Concurrent Constraint Sprachen . . . 217

12 Zusammenfassung und Ausblick 219

A Die virtuelle Maschine für L 225
A.1 Adressierungsarten . . . 225
A.2 Register . . . 226
A.3 Instruktionen . . . 227
A.3.1 Prelude . . . 227
A.3.2 Zellen, Konstruktoren, logische Variablen . . . 228
A.3.3 Tupel . . . 228
A.3.4 Unifikation . . . 228
A.3.5 Arithmetik . . . 229
A.3.6 Threads . . . 229
A.3.7 Funktionen . . . 229
A.3.8 Case . . . 230
A.3.9 Sonstige Instruktionen . . . 232

B Applikationen für Testläufe 233

C Benchmark-Programme 235
C.1 Takeushi . . . 235
C.1.1 Takeushi mit ganzen Zahlen . . . 235
C.1.2 Takeushi unter Verwendung von CPS . . . 235
C.2 Fibonacci . . . 236
C.2.1 Fibonacci mit ganzen Zahlen . . . 236

C.2.2 Fibonacci mit Fließkommazahlen . . . 236
C.3 Naiv Reverse . . . 236
C.4 Quicksort . . . 236
C.4.1 Quicksort mit Listen . . . 236
C.4.2 Quicksort mit Listen und higher-order Tests . . . 236
C.4.3 Quicksort mit Feldern . . . 237
C.5 N-Damen . . . 238
C.6 Mandelbrot . . . 238
C.7 Symbolische Ableitung . . . 239
C.8 Threads . . . 240
C.8.1 Erzeugung . . . 240
C.8.2 Kommunikation . . . 240
C.8.3 Fibonacci mit Threads . . . 241

Kapitel 1

Einleitung

Diese Arbeit beschreibt *Design*, *Implementierung* und *Evaluierung* einer virtuellen Maschine für den Kern der Programmiersprache Oz. Wir konzentrieren uns auf Teilaspekte der Implementierung, indem wir auf eine idealisierte Teilsprache von Oz zurückgreifen. Wichtige Charakteristika dieser Teilsprache sind: logische Variablen, Threads, Synchronisation und dynamische Typisierung.

Seit 1991 wird Oz am DFKI unter Leitung von Gert Smolka entwickelt. Der Autor war zusammen mit Michael Mehl für den Kern der Implementierung von Oz verantwortlich. Oz vereint Strukturen aus logischer, funktionaler und objektorientierter Programmierung in sich. Im Fokus stand von Anfang an die Entwicklung einer Implementierung der Sprache, deren Anwendungstauglichkeit über die eines Forschungsprototypen hinaus gehen sollte. Es wurde darauf Wert gelegt, ein System zu entwickeln, das auch den Anforderungen gewachsen war, die die Realisierung ernsthafter Anwendungen aus der wirklichen Welt stellt. So diente Oz dann später auch als Implementierungssprache für eine ganze Reihe von Projekten (z.B. [KFM94, AFH95, HW96, KFP95, HLZ96, Wal96, SS96, CH96, SW98]). Dadurch konnten der Wert und die Praktikabilität neuer Ideen zum Design der Sprache an der existierenden Implementierung und deren Applikationen überprüft und gegebenenfalls modifiziert oder verworfen werden.

Die Arbeiten an der Implementierung von Oz mündeten 1995 in der offiziellen Freigabe des Systems, das DFKI Oz genannt wurde, zunächst in der Version 1.0 [Oz95] und später dann 2.0 [Oz97]. Damit waren die Arbeiten an Oz aber keineswegs abgeschlossen: zur Zeit steht die Freigabe des Nachfolgesystems von DFKI Oz kurz bevor, das den Namen Mozart trägt [Moz98]. Mozart ist eine Weiterentwicklung von DFKI Oz, an der nun auch das Swedish Institute of Computer Science (SICS) und die Université catholique de Louvain in Belgien mitwirken. Die Änderungen in Mozart gegenüber DFKI Oz bestehen in einer Vielzahl von Neuerungen und Erweiterungen. So zum Beispiel einer völligen Neuimplementierung des Compilers, eines Modulsystems mit getrennter Übersetzung und dynamischem Binden und vor allem einer Erweiterung der Sprache um die Möglichkeit zur verteilten Programmierung [VHB$^+$97, HVS97, HVBS98], um nur die wichtigsten zu nennen.

Wir können in dieser Arbeit nicht auf alle Aspekte der Implementierung der vollen Sprache Oz eingehen. Wir konzentrieren uns daher auf eine interessante, nicht-triviale, idealisierte Teilsprache von Oz, die wir im folgenden als L bezeichnen werden. Wichtige Aspekte von Oz, die wir in L ausgeblendet haben, werden in eigenen Arbeiten ausführlich beleuchtet: objektorientierte Programmierung [Hen97], Records [Meh99, RMS96], lokale Berechnungsräume und Suche

[Sch99, MSS95] und Constraints über endlichen Bereichen [Wür98] sind orthogonale Erweiterungen von L.

Wir werden eine effiziente Implementierung von L beschreiben, die uneingeschränkt auf volles Oz übertragbar ist. Das heißt, wir werden keine speziellen Optimierungen diskutieren, die nur für L gültig sind. Wir konzentrieren uns auf die Beschreibung einer virtuellen Maschine für L; die Funktionsweise des Compilers ergibt sich dann in der Regel direkt aus dieser. Wir beschäftigen uns insbesondere auch nicht mit der Vorstellung fortgeschrittener Compilationstechniken; diese stehen auch in Mozart erst am Anfang ihrer Entwicklung.

Wir wollen den Leser nicht mit dem Erlernen einer neuen Sprache samt deren Syntax belasten. Für L verwenden wir daher die Syntax von SML [MTHM97] mit geringfügigen Variationen. Insbesondere entsprechen die Konstrukte, die wir in L aus SML übernommen haben (z.B. Referenzen, Tupel, höhere Funktionen), auch semantisch einander. Das erlaubt dem Leser einen schnellen Einstieg, indem er sich auf die Unterschiede der beiden Sprachen konzentrieren kann. Die wichtigsten Erweiterungen von L gegenüber SML sind: logische Variablen, Threads und Synchronisation.

1.1 Logische Variablen

Logische Variablen erlauben die Darstellung von unvollständiger Information im Speicher, nämlich die explizite Repräsentation der Tatsache, daß der Wert einer Variablen noch unbestimmt ist.

Durch den Einsatz von logischen Variablen wird eine natürliche Integration des nebenläufigen Programmierparadigmas ermöglicht [Sar93]. Daneben erlauben logische Variablen eine Reihe von eleganten und zugleich effizienten Programmiertechniken, wie zum Beispiel das Arbeiten mit Differenzlisten [O'K90]. Weiter können durch logische Variablen zustandsfreie zyklische Datenstrukturen erzeugt werden. Schließlich werden wir sehen, daß sich viele Funktionen zur Topdown-Konstruktion von Datenstrukturen (wichtige Beispiele sind etwa `append` und `map`) endrekursiv und damit effizienter unter Verwendung logischer Variablen formulieren lassen, was in einer rein funktionalen Sprache nicht direkt möglich ist.

Darüber hinaus bilden logischen Variablen die zentrale Datenstruktur zur Lösung von Suchproblemen, wie beispielsweise im Bereich des Constraint Logischen Programmierens (CLP) [DVS+88, CKC83]. Auf diesen Aspekt werden wir in dieser Arbeit aber nicht eingehen und verweisen im Zusammenhang mit Oz auf [Sch99] und [Wür98].

Ein Ziel dieser Arbeit ist es, aufzuzeigen, daß sich logische Variablen in einer virtuellen Maschine mit keinen oder nur sehr geringen Kosten für Speicherplatz und Laufzeit realisieren lassen.

1.2 Threads

Simula [DMN67] erlaubte als erste Programmiersprache mit Coroutining eine einfache Form nebenläufiger Berechnung. Mittlerweile verfügt fast jede neuere Programmiersprache über entsprechende Konstrukte oder Erweiterungen [AVWW96, GJS97, Nie97]. Dabei kommt es aber weniger auf die Möglichkeit an, mehrere Berechnungsfäden (Threads) abspalten zu können, viel wichtiger sind die Primitive, die eine möglichst einfache Kommunikation und Synchronisation

der Threads untereinander erlauben. Hier bietet die Verwendung logischer Variablen ein einfaches, transparentes (weil implizites) und zugleich mächtiges Mittel zur Synchronisation von Threads: ein Thread, der beispielsweise eine Addition $x + y$ ausführen will, hält einfach an, wenn etwa x noch auf eine logische Variable verweist. Sobald nun ein anderer Thread x an eine Zahl bindet, kann der erste Thread seine Arbeit wieder aufnehmen.

Wir trennen den Begriff Nebenläufigkeit klar von dem Begriff *Parallelität* ab. Unter Parallelität verstehen wir einen Implementierungaspekt, bei dem ein Programm zeitgleich auf verschiedenen Ausführungseinheiten abgearbeitet wird. Somit sind die Begriffe Nebenläufigkeit und Parallelität weitgehend unabhängig voneinander, da auch sequentielle Sprachen parallel ausgeführt werden können, wie auch nebenläufige Sprachen nicht unbedingt zum Zwecke der parallelen Ausführung entworfen worden sein müssen (was auch für Oz gilt, obgleich auch an einer parallelen Implementierung von Oz gearbeitet wird [Pop97]). In dieser Arbeit werden wir uns auf die Beschreibung einer sequentiellen Implementierung von Oz beschränken.

1.3 Synchronisation

Eine einfache Form der Synchronisation haben wir im vorangehenden Abschnitt am Beispiel einer Addition angesprochen. Wesentlich komplexere Formen der Synchronisation erlaubt in L die Verwendung eines Ausdrucks der Form

```
case x of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

Anders als in SML dürfen in L in den Wächtern (engl. guards) p_i freie Variablen auch mehrfach vorkommen. Die Semantik ist über logische Implikation (Subsumption) definiert, was eine besondere Schwierigkeit für die Implementierung darstellt: es gilt zu entscheiden, ob der Speicher die Gleichung $\exists\overline{y}\ (x = p_i)$ impliziert, wobei $\overline{y}$ die freien Variablen von p_i sind. Die Schwierigkeit liegt hier zum einen in der korrekten Behandlung des Existenzquantors, zum anderen in der Tatsache, daß freie Variablen mehrfach in p_i vorkommen dürfen. Darüber hinaus kann das Erfülltsein der Bedingung nicht immer sofort entschieden werden: wenn die Information im Speicher dazu noch nicht ausreichend ist, muß die Auswertung des Konstruktes zurückgestellt und zum richtigen Zeitpunkt wieder aufgenommen werden.

Das Konstrukt geht zurück auf die Familie der committed-choice Sprachen [Mah87] und das nebenläufige Constraint-Programmieren (CCP) [Sar93]. Vertreter dieser Sprachen [Ued85, Sha87, UC90, Con89] und deren Implementierungen waren aber entweder ineffizient oder schränkten die Art und Verwendung von Wächtern unter Verletzung der Semantik ad hoc ein (z.B. read-only Variablen, modes).

Wir werden in dieser Arbeit eine korrekte und zugleich effiziente Implementierung von flachen Wächtern mit Subsumptions-Semantik vorstellen.

1.4 Schnelle Emulatoren

Die etablierte Technik zur Implementierung von Programmiersprachen besteht im Entwurf einer *virtuellen* Maschine [WM97]. Im Gegensatz zu einer *realen* Maschine handelt es sich dabei um die idealisierte Darstellung einer Architektur, die keine real existierende Hardware beschreibt. Um den Code einer virtuellen Maschine auf einer realen Zielplattform zur Ausführung zu bringen, gibt es verschieden Möglichkeiten: man kann beispielsweise in einem zusätzlichen Übersetzungsschritt realen Maschinen- oder Assemblercode für die Zielplattform erzeugen. Man kann aber auch den virtuellen Maschinencode in eine andere maschinennähere Hochsprache (zum Beispiel C) übersetzen. Schließlich gibt es noch die Möglichkeit, den virtuellen Maschinencode durch Implementierung eines Interpreters zur Ausführung zu bringen. Bei den ersten beiden Möglichkeiten spricht man im allgemeinen von der Erzeugung von *nativem* Code. Dagegen bezeichnet man einen Interpreter auch als *Emulator*, eine Technik, die wir auch bei der Implementierung von DFKI Oz und Mozart verwandt haben.

Der wesentliche Vorteil eines Emulators besteht in seiner hohen *Portabilität*. So benötigt nach unseren Erfahrungen die Portierung des gesamtem Mozart Systems auf eine neue UNIX Plattform in der Regel nur wenige Stunden. Auch die Portierung nach MS Windows unterstrich die Plattformunabhängigkeit der Implementierung des Emulatorkerns und der damit verbundenen Datenstrukturen. Diejenigen Teile, die hier umfangreichere Anpassungen mit sich zogen (wie Signalbehandlung, Ein-/Ausgabe oder Interprozesskommunikation), waren dagegen davon unabhängig.

Eng damit verbunden liegt ein weiterer Vorteil eines Emulators in der Flexibilität und der Offenheit für Experimente, mit der man auf die Anforderungen einer in der Entwicklung befindlichen Programmiersprache reagieren kann. Oz wäre sicher nicht da, wo es heute ist, hätte man gleich auf eine andere Implementierungstechnik zurückgegriffen.

Ein wichtiger Faktor für die Akzeptanz durch die Benutzer ist die Performanz eines Systems. Um bestehen zu können, darf weder dessen Speicherverbrauch noch vor allem dessen Laufzeitverhalten deutlich hinter vergleichbaren Systemen zurückfallen. Dagegen nimmt der Benutzer gewisse Performanzeinbußen in Kauf, wenn er zum Ausgleich mit einer größeren Expressivität der Sprache belohnt wird. Zu Projektbeginn schien unter Berücksichtigung der Ausdruckskraft des ersten Sprachdesigns und der Performanz von bereits existierenden vergleichbaren Systemen ein Fernziel zwar ambitioniert aber nicht unrealistisch: Oz sollte bis auf einen Faktor von zwei bis drei an die Performanz von effizienten Prolog Implementierungen wie Sicstus oder Quintus Prolog herankommen (Prolog wurde als Maßstab herangezogen, da Oz in der Anfangsphase noch stark vom logischen Programmieren beeinflußt war).

Emulatoren stehen im allgemeinen in dem Ruf, eine nur mäßige Ausführungsgeschwindigkeit von Programmen zu erlauben. So fallen diese in der Regel bei Anwendungen, bei denen numerische Berechnungen dominieren, um eine Größenordnung hinter Systemen zurück, die nativen Code erzeugen. Für solche Anwendungen wurde Oz aber nicht entworfen. Wir werden in dieser Arbeit sehen, daß die Unterschiede für die eigentlichen Anwendungsgebiete von Oz, bei denen symbolische Berechnungen im Vordergrund stehen, deutlich geringer ausfallen. So werden wir unter anderem Techniken zur Konstruktion hoch-performanter Emulatoren vorstellen, die in dieser Disziplin in der Regel auf einen Faktor von zwei bis drei an sehr gute native Systeme herankommen und in bestimmten Fällen sogar leicht besser sein können.

Im Vergleich mit anderen Emulatoren nimmt Mozart eine Spitzenstellung ein. Wir werden sehen,

daß Mozart kompetitiv ist mit den schnellsten Emulatoren für statisch getypte Sprachen. Mozart ist in der Regel etwa doppelt so schnell wie Sicstus Prolog und Erlang, Sprachen, die von ihrem Charakter her (dynamische Typisierung, Nebenläufigkeit, logische Variablen) eher dem von Oz entsprechen als etwa die ML Dialekte. Somit haben wir unser ursprüngliches Ziel sogar deutlich übertroffen.

Dennoch ist die Implementierung von Oz nicht kompromißlos auf höchste Performanz ausgerichtet. Eines der wichtigsten Kriterien für die Implementierung war stets die Auslegung auf den praktischen Einsatz. Demzufolge mußten häufig Kompromisse eingegangen werden. Das heißt Effizienz mußte vielfach hinter andere Aspekte zurücktreten, wie beispielsweise Wartbarkeit des Systems, Offenheit für neue Ideen, Größe des erzeugten Codes, Laufzeit des Compilers oder Speicherplatzverbrauch. Zudem umfaßt der Sprachumfang von Oz viele Aspekte aus den unterschiedlichsten Programmierparadigmen und viele vordefinierte Datentypen, was Optimierungen erschwert. So kann einerseits die gleichzeitige Optimierung aller Aspekte schnell die Komplexität der Implementierung derart erhöhen, daß diese nicht mehr beherrschbar wird. Zum anderen ist die Optimierung eines Aspektes oft nur mit einer Verschlechterung an anderer Stelle erreichbar.

1.5 Beiträge

Wir stellen in dieser Arbeit eine informelle Beschreibung eines **Graphenmodells** vor, das zur Definition der dynamischen Semantik von L dient. Das Graphenmodell setzt auf einem niedrigeren Abstraktionsniveau an als die Sprachdefinitionen *The Oz Programming Model* [Smo95b] oder *The Definition of Kernel Oz* [Smo95a], die die Sprache über das Constraintmodell definieren. Das Graphenmodell bietet zwei wesentliche Vorteile: zum einen stellt es einen klaren Ausgangspunkt dar, auf dessen Basis sich eine effiziente Implementierung der Sprache realisieren läßt. Daher gibt es zum anderen auch erstmals ein *Kostenmodell* in Platz und Zeit für Oz, das Aussagen sowohl über Speicher- als auch Laufzeitverhalten von Programmen erlaubt.

Wir stellen eine **effiziente Implementierung der Synchronisation** mittels `case` vor. Existierende Implementierungen vergleichbarer Konstrukte sind entweder ineffizient oder schränken die Art und Verwendung von Wächtern ad hoc ein. Wir zeigen zudem, daß sich in vielen praktisch relevanten Fällen die Wächter eines Case genauso effizient übersetzen lassen, wie dies etwa in funktionalen Sprachen geschehen kann.

Wir stellen eine Implementierung für besonders **leichtgewichtige Threads** vor: die Kosten für die Erzeugung eines Threads liegen mehr als eine Größenordnung unter den Kosten für die gleiche Operation in Java. Die Existenz von Threads bedeutet keine Beeinträchtigung der Performanz von sequentiellen Programmen. Im Zusammenhang mit Threads beschreiben wir eine flexible *Nonstandard-Darstellung des Kellers* eines Threads, die von den gängigen Implementierungstechniken deutlich abweicht: dies erlaubt die flexible Darstellung beliebiger heterogener Information auf dem Keller, ohne nennenswerte Einbußen mit sich zu bringen.

Die Arbeit liefert Beiträge zur effizienten Implementierung von Programmiersprachen in den Bereichen logische, funktionale, nebenläufige und Constraint-Programmierung. Wir belegen in dieser Arbeit, daß die existierende Implementierung Mozart **kompetitiv mit den besten Emulatoren** für statisch getypte funktionale Sprachen (z.B. OCAML) ist.

Wir stellen eine Implementierungstechnik, die **dynamische Codespezialisierung**, vor. Diese Technik erlaubt es unter anderem, einige der Nachteile (aus Implementierungssicht) von dynamisch getypten Sprachen gegenüber statisch getypten Sprachen wett zu machen. Sie beruht auf

der Tatsache, daß sich die Werte globaler Variablen dynamisch nach Konstruktion nicht mehr ändern können. Davon ausgehend werden dann dynamisch zur Laufzeit Optimierungen am Maschinencode vorgenommen.

Eine Anwendung der dynamischen Codespezialisierung ist die **effiziente Implementierung von höheren Funktionen in einer dynamisch getypten Sprache**. Der Einsatz der Technik erlaubt es beispielsweise, viele *Funktionsaufrufe* zu optimieren: dadurch brauchen dann zur Laufzeit keine Dereferenzierungsschritte, Typ- und Konsistenztests mehr durchgeführt zu werden. Weiter kann die Größe von *Funktionsabschlüssen* deutlich reduziert werden: da Referenzen auf viele globale Bezeichner direkt durch Referenzen im Code aufgelöst werden, müssen diese nicht mehr in die dynamisch erzeugten Funktionsabschlüsse aufgenommen werden. Das Vorgehen ist besonders effektiv, da beispielsweise bei den meisten Funktionaufrufen die Funktion über einen globalen Bezeichner angesprochen wird.

Eine weitere Anwendung der dynamischen Codespezialisierung ist die **effiziente Implementierung von Namen**: Namen sind ein zentrales Sprachmittel von Oz zum dedizierten Verbergen und Sichtbarmachen von Informationen (z.B für private Methoden und Attribute von Objekten). Das geschieht dadurch, daß Namen nur indirekt über Variablen referiert werden können. In einer dynamisch getypten Sprache gehen dadurch allerdings im allgemeinen alle Optimierungsmöglichkeiten verloren. Durch Codespezialisierung kann dies in der Praxis in fast allen Fällen wieder wett gemacht werden, so daß Namen hier genauso effizient gehandhabt werden können, wie anderen Konstanten (z.B. Zahlen oder Atome) auch.

Wir beschreiben eine Technik zur effizienten Implementierung von **Funktoren**. Funktoren sind Funktionen, die Module erzeugen. Eingabeparameter dieser Funktionen sind diejenigen Module, die vom zu erzeugenden Modul verwandt werden. Diese Konstruktion führt allerdings dazu, daß der Compiler deutlich weniger modul-interne und -externe Optimierungen vornehmen kann. Wir zeigen daher, daß man durch *Kopieren des Codes* des Rumpfes eines Funktors bei jeder Applikation des Funktors dieses Manko wieder ausgleichen kann: der Code jeder Kopie kann wieder in Abhängigkeit von den Werten der Eingabeparameter, globaler und lokaler Variablen optimiert werden.

Wir stellen ein **flexibles Format zur Darstellung von Maschinenprogrammen** vor. Anhand dieses Formats läßt sich einfach die Vorgehensweise zur Realisierung einer interaktiven Entwicklungsumgebung und zum Export und Import von Bezeichnern in einem separat übersetzten Modul erklären. Wir zeigen weiter, daß dieses Format leistungsfähig und leicht erweiterbar ist, so daß damit die Erfordernisse für *Persistenz* und *Verteilung* realisiert werden können.

Die Gesamtperformanz eines Systems hängt auch von vielen Details auf unterer Implementierungsebene ab. Wir stellen in dieser Arbeit eine Reihe solcher **Implementierungstechniken** vor. Diese sind zum Teil neu, zum Teil haben wir auch Techniken zusammengetragen, die zwar nicht neu, aber dennoch nach unseren Erfahrungen nur wenig bekannt sind, so daß sich jeder Implementeur diese wieder aufs neue selbst erschließen muß. Dabei handelt es sich einerseits um Techniken, die speziell zur Implementierung hoch performanter Emulatoren benötigt werden. Wir werden aber auch allgemein einsetzbare Verfahren vorstellen, wie zum Beispiel ein ausgefeiltes Schema zur Darstellung von markierten Referenzen oder eine einfache aber dennoch in der Praxis wirkungsvolle und effiziente Freispeicherverwaltung.

Die Arbeit wird angereichert von **Zahlenmaterial aus realen Anwendungen**. Diese Zahlen geben Aufschluß über das dynamische Verhalten von großen Applikationen, wie zum Beispiel den Anteil verschiedener Datenstrukturen am Speicherverbrauch, die Art und Länge von Referenz-

ketten oder die Verwendung von Unifikation und Threads. Aufgrund dieser Zahlen kann man dann beispielsweise erkennen, welche Optimierungen besonders wirkungsvoll und welche weniger erfolgversprechend sind; auch Schwachpunkte der aktuellen Implementierung lassen sich dadurch bestimmen.

1.6 Gliederung der Arbeit

Um der Fülle des Materials Herr zu werden, nehmen wir eine Strukturierung in unterschiedliche *Abstraktionsebenen* vor und verzichten dem Charakter einer praktischen Arbeit entsprechend auf eine streng formale Darstellung. Wir konzentrieren uns vielmehr darauf, Ideen und Prinzipien informell zu vermitteln. So verzichten wir beispielsweise sowohl auf die Formulierung von Theoremen einerseits als auch auf die detaillierte Darstellung der Instruktionen der virtuellen Maschine etwa durch Verwendung von C-Code andererseits. Dennoch sollte es auch dem weniger erfahrenen Leser möglich sein, anhand dieser Arbeit eine Implementierung von L selbst nachzuvollziehen.

Die Arbeit besteht aus vier Teilen: *Sprachbeschreibung*, *virtuelle Maschine*, *Implementierung* und *Evaluierung*. Wir nehmen dadurch eine Strukturierung in unterschiedlichen Abstraktionsstufen vor, die wir auch innerhalb der einzelnen Teile weiter fortführen. So können wir wichtige Ideen jeweils auf einem möglichst hohem Abstraktionsniveau vermitteln. Zudem erlaubt dies ein selektives Lesen: dem Leser der nur an den wesentlichen Konstruktionsprinzipien der Maschine aus abstrakter Sicht gelegen ist, reicht ein Lesen der ersten beiden Teile, die zudem keine tiefergehenden Kenntnisse der Materie voraussetzen. Dagegen kann derjenige der sich auch für Aspekte der Implementierung interessiert, entsprechend weiter vordringen. Am Ende jedes Kapitels geben wir jeweils kurze, stichpunktartige Zusammenfassungen an.

In Teil I führen wir die Sprache L ein. Kapitel 2 definiert die Sprache informell anhand des Graphenmodells und gibt im zweiten Teil des Kapitels eine Reihe von Beispielen, die die Verwendung der Konstrukte der Sprache verdeutlichen.

Teil II beschreibt das Design einer virtuellen Maschine für L. In Kapitel 3 geben wir zunächst ein einfaches idealisiertes Modell der Maschine an. Anschließend gehen wir in Kapitel 4 genauer auf mögliche Optimierungen der Maschine ein. In Kapitel 5 besprechen wir das für und wider verschiedener Designentscheidungen und diskutieren alternative Möglichkeiten.

Im dritten Teil der Arbeit gehen wir auf die Implementierung des Modells aus Teil II ein. Wir beginnen mit einer Beschreibung des Emulators in Kapitel 6 und gehen in Kapitel 7 auf das Speichermanagement und die Darstellung von Datenstrukturen im Speicher ein. In Kapitel 8 gehen wir auf Verfeinerungen bei der Implementierung von Threads gegenüber dem Modell aus Teil II ein. Kapitel 9 zeigt, wie Funktionen aus anderen maschinennäheren Sprachen in L transparent verfügbar gemacht werden können, was auch die Möglichkeit der Suspension einschließt.

Im letzten Teil der Arbeit vergleichen wir in Kapitel 10 Mozart mit anderen Systemen anhand verschiedener gängiger Benchmarks und untersuchen das dynamische Verhalten größerer Oz Applikationen. In Kapitel 11 schließlich ziehen wir Vergleiche mit verwandten Arbeiten.

Teil I

Die Sprache

Kapitel 2

Die Sprache L

In diesem Kapitel werden wir die Sprache L einführen und eine informelle Beschreibung der dynamischen Semantik von L angeben. Bei L handelt es sich um eine echte Teilsprache von Oz, mit einer einzigen Ausnahme: die Sprachdefinition von L enthält echte Funktionen, während Oz lediglich relationale Prozeduren anbietet. Diese Erweiterung hat aber keinen wesentlichen Einfluß auf die Sprachdefinition und bedingt nur minimale Änderungen aus Sicht der Implementierung, so daß prinzipiell das sogenannte *Constraintmodell* aus [Smo95b, Smo95a] direkt übertragen werden kann. Wegen seines mathematisch abstrakten Charakters hat dieses Modell den Vorteil, daß sich damit die Semantik der Sprache exakt und kompakt beschreiben läßt. Aus praktischer Sicht jedoch gibt dieses Modell kaum Anhaltspunkte für eine Implementierung der Sprache, die den Kern dieser Arbeit bildet: so wird beispielsweise bei jeder Applikation einer Prozedur eine vollständige *Kopie* des Rumpfes der Prozedur angefertigt. Eine direkte Umsetzung dieses Modells ist daher aus praktischer Sicht nicht geboten.

Für den Benutzer einer Programmiersprache ist es auch nicht ausreichend, nur zu wissen, daß die Implementierung die semantischen Vorgaben einhält. Vielmehr muß es dem Anwender auch möglich sein, ein *Performanzmodell* seiner Programme entwickeln zu können: so sollte das Berechnungsmodell genaue Aussagen sowohl über den Speicherverbrauch als auch über die Laufzeit eines Programmes erlauben. Wir werden daher zunächst in diesem Kapitel das Graphenmodell von L einführen. Dieses werden wir dann im folgenden Kapitel 3 schließlich noch einmal weiter zum Maschinenmodell detailieren, das dann schon sehr dicht auf der konkreten Implementierung von Mozart aufsetzt.

Wir werden uns an dieser Stelle auf eine möglichst kompakte und eher informelle Darstellung beschränken, die auf die Erfordernisse der Beschreibung der Implementierung ausgerichtet ist, die Gegenstand dieser Arbeit ist. Den interessierten Leser verweisen wir auf die Arbeiten [Smo95a] und [Smo95b], die eine Definition von Oz 1 anhand des Constraintmodells geben und auf [Smo98], das eine formalere Darstellung aus Sicht der funktionalen Welt gibt.

Wir beginnen das Kapitel mit den Abschnitten 2.1 und 2.2 mit einem Überblick über L. Wir beschreiben L, indem wir zunächst eine Teilsprache namens L– betrachten. In den Abschnitten 2.4 bis 2.9 geben wir eine informelle Beschreibung der Semantik von L–. Danach betrachten wir dann die Erweiterungen die von L– nach L führen: in Abschnitt 2.10 gehen wir auf die Integration logischer Variablen in L ein und beschreiben in Abschnitt 2.11 Nebenläufigkeit in L durch die Einführung von Threads. Wir schließen das Kapitel ab, indem wir in Abschnitt 2.13 anhand verschiedener Beispiele die Verwendung interessanter Konstrukte der Sprache (logische

Abbildung 2.1 Die Sprache L

e	::=	x	*Variable*
	|	n	*Ganze Zahl*
	|	$funexp$	*Funktion*
	|	$e_0(e_1, \ldots, e_n)$	*Applikation*
	|	**case** e **of** $mrule_1 \mid \ldots \mid mrule_n$	*Case*
	|	**let** d **in** e **end**	*Deklaration*
$funexp$	::=	**fn** $(x_1, \ldots, x_n)$ **=>** e	*Funktion*
d	::=	**val** x **=** e	*Variablendeklaration*
	|	**con** x	*Konstruktordeklaration*
	|	**rec** $x = funexp$	*rekursive Funktion*
$mrule$	::=	p **=>** e	*Match*
p	::=	$x \mid n$	
	|	$x(p_1, \ldots, p_n)$	*Tupel*

Variablen, Threads, Konstruktoren) verdeutlichen.

2.1 Überblick über die Sprache L

Wie eingangs erwähnt, verwenden wir die Sprache L als Vehikel zur Beschreibung wichtiger Aspekte der Implementierung von Oz. Wir wollen den Leser möglichst wenig mit dem Erlernen einer neuen Sprache samt deren Syntax belasten. Zur Definition von L verwenden wir daher Konstrukte aus der Syntax von SML [MTHM97] mit geringfügigen Variationen, die auch semantisch einander entsprechen (vgl. Abbildung 2.1). Wir setzen daher Grund-Kenntnisse zu SML voraus und erlauben dem Leser dadurch einen schnellen Einstieg, indem er sich auf die Unterschiede der beiden Sprachen konzentrieren kann. Wir ziehen SML gegenüber Scheme vor, da SML die reicheren Datenstrukturen bietet und Zellen explizit herausfaktorisiert.

Erweiterungen gegenüber SML werden in L über Operationen (vgl. Abbildung 2.2) eingeführt. Das sind vordefinierte Funktionen, auf deren Bedeutung wir erst später eingehen werden.

Wir geben an dieser Stelle bereits einen Überblick über L, indem wir kurz die wichtigsten Unterschiede und Gemeinsamkeiten zu SML benennen:

- L erlaubt die Verwendung logischer Variablen.
- L ist nebenläufig.
- Das `case`-Konstrukt von L ist ausdrucksstärker als das gleiche Konstrukt in SML: seine Semantik ist über logische Implikation definiert; zudem sind in L nicht-lineare Patterns erlaubt.

Abbildung 2.2 Operationen von L (`lvar`, `unify` und `spawn` nur in L nicht in L–)

```
+,-,*,/ : int * int -> int
<       : int * int -> bool
ref     : α -> α ref
exchange: α ref * α -> α


lvar    : unit -> α
unify   : α * α -> unit
spawn   : (unit -> α) -> unit
```

- L ist dynamisch getypt.
- L übernimmt einen Teil der Datenstrukturen aus SML: ganze Zahlen, Zellen (die in SML als Referenzen bezeichnet werden), höhere Funktionen und Tupel.
- L erlaubt im Gegensatz zu SML die Konstruktion zyklischer Tupel (rationale Bäume).
- Während SML nur einstellige Funktionen kennt, erlaubt L die direkte Definition mehrstelliger Funktionen.

2.2 Notation

Um die Formulierung von Beispielprogrammen zu erleichtern, definieren wir in Abbildung 2.3 verschiedene abgeleitete Formen. Dabei handelt es sich lediglich um syntaktischen Zucker, deren Semantik über die in Abbildung 2.3 angegebenen äquivalenten Formen definiert ist.

Variablen schreiben wir wie in [MTHM97] festgelegt, verwenden also eine Folge von Buchstaben und Ziffern oder eine Folge von Sonderzeichen. Beispiele für Variablennamen sind also etwa:

Abbildung 2.3 Abgeleitete Formen

Abgeleitete Form	Äquivalente Form
e_1 ; e_2	**let** $x = e_1$ **in** e_2 **end**
let $d_1; \ldots; d_n$ **in** e **end**	**let** d_1 **in** ... **let** d_n **in** e **end end**
fun $x(y_1, \ldots, y_n) = e_1; e_2$	**let rec** $x =$ **fn** $(y_1, \ldots, y_n)$ **=>** e_1 **in** e_2 **end**
$[e_1, e_2, \ldots, e_n]$	e_1::e_2,...::e_n::nil
thread e	**let** x=lvar() **in** spawn(fn()=>unify(x,e)); x **end**

```
x  y25  aux  Aux  append  MakeList  _  ::  +$$::@@
```

Um Variablen, die auf Konstruktoren[1] verweisen, syntaktisch leicht von anderen Variablen unterscheiden zu können und dadurch die Lesbarkeit von Programmen zu erhöhen, verwenden wir eine Schreibweise, die sich auch in SML eingebürgert hat [Pau96]: wir beginnen solche Variablen mit einem Großbuchstaben:

```
Tree  Node  Red  Green
```

Wir werden im folgenden häufiger Listen verwenden, für die wir wie in SML den Listenkonstruktor `::` in Infixschreibweise verwenden, die leere Liste schreiben wir als `nil`. Wie aus Abbildung 2.3 hervorgeht, verwenden wir als Abkürzung für längere Listen auch

`[1,2,3,4]` statt `1::2::3::4::nil`.

und schreiben insbesondere die leere Liste auch als `[]`.

Weiter verwenden wir **`true`** und **`false`** zur Darstellung der Booleschen Werte in L und schreiben den Einheitskonstruktor als **`unit`** (an Stelle von `()` in SML).

Wir gehen also davon aus, daß ein auszuwertendes Programm e stets automatisch in

```
let con ::;
    con nil;
    con true;
    con false;
    con unit
in
   e
end
```

eingeschlossen wird, wobei wir erst weiter unten auf die Semantik der Konstruktordeklaration eingehen werden.

2.3 Die Sprachen L– und L

Wir werden die Sprache L im folgenden in zwei Schritten vorstellen, indem wir zunächst die Sprache L– beschreiben werden, und danach erst auf L eingehen. Bei L– handelt es sich um eine Teilsprache von L mit den folgenden Einschränkungen gegenüber L:

- L– ist nicht nebenläufig.
- L– enthält keine logischen Variablen.
- Die Konstruktion zyklischer Tupel ist in L– nicht möglich (wegen des Fehlens logischer Variablen)

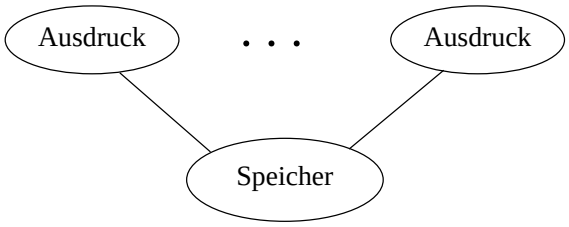
[1] Wir werden erst weiter unten genauer definieren, was wir in L unter einem Konstruktor verstehen.

- L– erlaubt nur lineare Pattern im Case.

Somit ist L– auch eine Teilsprache von SML (bis auf die in L– zur Konvenienz eingeführten mehrstelligen Funktionen). Wir müssen daher die Bedeutung der einzelnen Sprachkonstrukte von L– im folgenden nicht mehr im Detail motivieren, sondern können uns auf die Beschreibung des Graphenmodells konzentrieren, anhand dessen wir die dynamische Semantik der Sprache informell beschreiben werden.

2.4 Der Speicher als Graph

Berechnung in L bedeutet Auswertung (oder *Evaluierung*) von *Ausdrücken* über dem *Speicher*:



Der Speicher wird als endlicher Graph mit markierten Knoten und gerichteten, unmarkierten Kanten dargestellt. Die Kanten, die von jedem Knoten wegführen, sind geordnet, das heißt, es ist klar definiert, welcher Knoten der erste, zweite, usw. Sohn eines Knotens ist.[2] Der Graph kann auch Zyklen enthalten: so werden wir weiter unten sehen, daß beispielsweise ein Knoten, der zur Darstellung einer Funktion dient, auf sich selbst verweisen kann, wenn die Funktion sich selbst rekursiv aufruft. Zudem werden wir sehen, daß in L gegenüber L– auch Zyklen anderer Art möglich sind: durch Einführung logischer Variablen und anschließendes Binden dieser Variablen können zyklische Tupel erzeugt werden.

Bei der Auswertung von Ausdrücken werden diese durch neue ersetzt (vgl. Abschnitt 2.8) und verändern in der Regel als Seiteneffekt den Inhalt des Speichers, worauf wir in Abschnitt 2.8 näher eingehen werden.

2.5 Werte und Knoten

Bei der Definition der Semantik von SML [MTHM97] und Oz [Smo95b, Smo95a] stehen Variablen als Platzhalter für mathematische Strukturen, die als *Werte* (engl. *value*) bezeichnet werden, die wiederum das *Universum* bilden. In unserer informellen Beschreibung des Graphenmodells stehen Variablen für *Knoten* im Graphen ([Smo98] verwendet dafür den Begriff *unit*). Bei der Abgrenzung der Begriffe Wert und Knoten sind zwei Punkte besonders zu beachten: zum einen dienen die Knoten zur Repräsentation von Werten, allerdings können unterschiedliche Knoten im Graphen den gleichen Wert beschreiben. Wir werden daher sehen, daß im Graphenmodell die Gleichheit von Knoten nicht über deren Identität definiert wird, vielmehr handelt es sich dabei um eine strukturelle Definition der Gleichheit, so daß genau solche Knoten als gleich definiert

[2] Dies kann man natürlich auch so auffassen, daß die Kanten ebenfalls markiert sind und zwar mit den Zahlen $1, 2, \ldots$

sind, die auch die gleichen Werte beschreiben. Der zweite Punkt bei der Abgrenzung der Begriffe betrifft die Rolle der logischen Variablen, auf die wir beim Übergang von L– zu L zurückkommen werden: bei einem Knoten im Graph kann es sich auch um eine logische Variable handeln. Damit wird explizit gemacht, daß der Wert einer Variable noch unbestimmt ist. Im Constraintmodell für Oz entspricht das gerade der Situation, daß der Speicher zu einer Variable keine Information enthält (vgl. [Smo95b]). In SML dagegen kann diese Situation nicht eintreten, da hier jede Variable gleich bei ihrer Definition einen Wert erhält.

Variablen stehen also für Knoten im Graphen; wir werden im folgenden aber auch vom Wert einer Variablen sprechen und meinen damit den Wert, den der entsprechende Knoten repräsentiert. Wie bereits erwähnt sind die Knoten des Graphen markiert. Da L dynamisch getypt ist, geben die Marken nicht nur Aufschluß über den Wert eines Knotens sondern auch über dessen Typ. Alle Knoten sind daher mindestens mit dem Typ des Wertes markiert, der eventuell noch mit weiterer Information versehen sein kann.

Im folgenden beschreiben wir die Knoten und die diesen entsprechenden Werte von L– (vgl. auch Abbildung 2.4 auf Seite 17):

Ganze Zahlen sind wie aus der Mathematik bekannt definiert. Im Graphen werden Zahlen als Knoten dargestellt, die mit der Marke `INT` und zusätzlich dem Wert n der Zahl markiert sind.

Zellen in L entsprechen genau den Referenzen in SML. Zustand ist ein unverzichtbares Konstrukt der Sprache im Hinblick auf Nebenläufigkeit und zur Realisierung von objektorientierter Programmierung [Hen97]. Ein Knoten, der eine Zelle darstellt, ist mit `CELL` markiert; von einem solchen Knoten führt genau eine Kante weg, die auf den Inhalt der Zelle verweist.

Konstruktoren sind primitive Werte ohne Struktur. Es gibt unendlich viele Konstruktoren, dargestellt als Knoten mit Marke `CON`. Gleiche Konstruktoren sind durch identische Knoten im Graphen repräsentiert. Konstruktoren in L entsprechen genau den Namen in Oz.

Konstruktoren in L unterscheiden sich insofern von den Konstruktoren in SML, als es sich bei letzteren um Funktionen (zur Erzeugung der Werte benutzerdefinierter Datentypen) handelt.

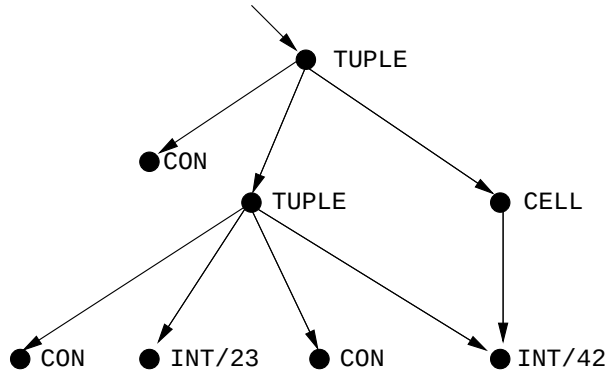
Tupel in L entsprechen den Tupeln aus SML. Für die Tupel in L muß allerdings zusätzlich immer gelten, daß es sich bei dem ersten Element um einen Konstruktor handelt, den wir als *Marke* des Tupels bezeichnen. Demzufolge benutzen wir in L auch eine andere Schreibweise und schreiben $f(x_1, \ldots, x_n)$ für ein Tupel mit Marke f und nicht wie in SML $(f, x_1, \ldots, x_n)$. Wegen des fehlenden statischen Typsystems in L können Typfehler durch geeignete Wahl von Marken für Tupel leichter dynamisch erkannt werden.

Ein Tupel der Form $f(x_1, \ldots, x_n)$ wird im Graphen durch einen Knoten dargestellt, der mit `TUPLE` markiert ist und von dem $n + 1$ Kanten wegführen. Die erste Kante verweist auf den Knoten für die Marke f, die restlichen n Kanten verweisen auf die sogenannten *Komponenten* x_i des Tupels. Wir bezeichnen n als die *Arität* eines Tupels.

Aus praktischer Sicht werden die Konstruktoren in L somit ganz ähnlich wie die Konstruktoren in SML verwandt, obgleich es sich dabei um zwei grundverschiedene Datentypen handelt. So kann insbesondere ein Konstruktor in L zur Konstruktion verschiedener Tupel mit unterschiedlicher Arität benutzt werden.

Funktionen sind ein integraler Bestandteil der Sprache. L unterstützt Funktionen als Datenstrukturen erster Ordnung, das heißt Funktionen können dynamisch erzeugt werden und es ist möglich, daß eine Variable x auch eine Funktion als Wert annehmen kann. Im Gegensatz zu SML, das nur einstellige Funktionen kennt, erlaubt L die Verwendung mehrstelliger Funktionen; wir kommen

Abbildung 2.4 Ausschnitt eines Graphen und ein Programm zu dessen Erzeugung



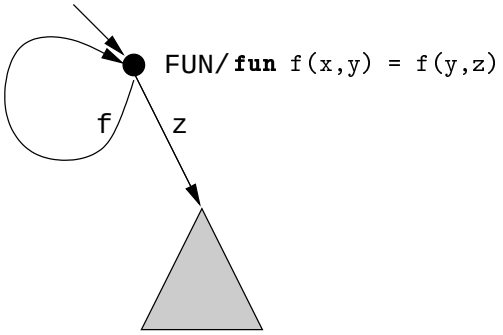

```
let val x = 42;
    val y = ref(x);
    con G;
    con ABC;
    con F;
in
    F(G(23,ABC,x),y)
end
```

darauf in Abschnitt 2.8 zurück.

Auch Funktionen sind durch Knoten im Graphen repräsentiert. Die Knoten werden dabei mit `FUN` und zusätzlich mit dem die Funktion definierenden Ausdruck markiert. Von einem Funktionsknoten führen auch Kanten weg, die auf die Werte der freien Variablen des Rumpfes verweisen. Eine Funktion, die mittels

```
fun f(x,y) = f(y,z)
```

definiert wurde, wird dann im Graphen wie folgt dargestellt:[3]



Man beachte, daß zu den freien Variablen des Rumpfes von `f` neben `z` auch `f` selbst gehört.

Beispiel

Abbildung 2.4 zeigt als Beispiel links einen Ausschnitt eines Ausführungsgraphen mit verschiedenen Knoten. Rechts in der Abbildung befindet sich ein Programm, das zur Erzeugung dieses Graphen dient. Wir werden allerdings erst in Abschnitt 2.8 auf die Auswertung von Ausdrücken eingehen.

[3] Wir haben der Übersichtlichkeit halber die Kanten mit den Namen der globalen Variablen beschriftet, obgleich wir zuvor festgelegt haben, daß die Kanten unmarkiert sein sollen. Genauer müßte man für jede globale Variable g je zwei Kanten vorsehen, wobei eine auf den Namen von g und eine auf deren Wert verweist; das wäre aber zu unübersichtlich.

2.6 Gleichheit

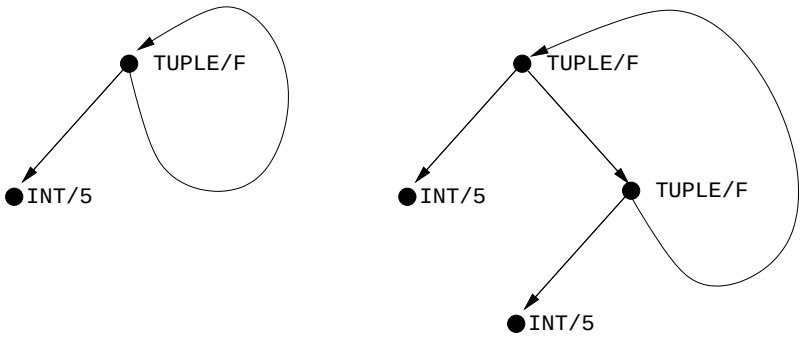
In diesem Abschnitt legen wir fest, wann zwei Knoten gleich sind, das heißt wir definieren eine Äquivalenzrelation $\sim$ auf Knoten.

$K \sim K'$ gilt genau dann, wenn eine der folgenden Bedingungen erfüllt ist:

- $K = K'$.
- K und K' sind beides Zahl-Knoten, die mit der selben Zahl markiert sind.
- K und K' sind beides Tupel-Knoten, für die gilt:
  - für die beiden Marken c respektive c' gilt $c \sim c'$.
  - für die Aritäten n respektive n' gilt: $n = n'$.
  - für die i-te Komponente K_i von K und K'_i von K' gilt: $K_i \sim K'_i$ für alle $1 \leq i \leq n$.

Die obige Definition von $\sim$ ist nun zwar für L–, nicht aber für L eindeutig: in L gibt es verschiedene Relationen, die die Bedingungen erfüllen, was daran liegt, daß in L Tupel Zyklen enthalten können. Wir definieren daher $\sim$ als die eindeutig definierte *größte* Relation, die die obigen Bedingungen erfüllt (in der kleinsten Relation sind beispielsweise Tupel mit Zyklen immer verschieden).

Als Beispiel betrachten wir die folgenden beiden Datenstrukturen



die gemäß Definition von $\sim$ gleich sind (wir haben der Übersichtlichkeit halber den `CON`-Knoten `F` für die Marke direkt an die Tupel-Knoten geschrieben), obwohl sich ihre Darstellung im Speicher unterscheidet.

Man beachte, daß inbesondere Konstruktoren, Zellen und Funktionen nur genau dann gleich sind, wenn sie durch den selben Knoten im Speicher dargestellt sind (token equality).

2.7 Umgebungen und Abschlüsse

Bei der Auswertung eines Ausdrucks e muß darauf geachtet werden, daß e auch freie Variablen enthalten kann. Eine gängige Lösung hierzu ist die, daß man in e keine freien Variablen zuläßt, sondern jede freie Variable in e direkt durch ihren Wert ersetzt. Das hat dann allerdings den Nachteil, daß etwa bei jedem Funktionsaufruf immer eine Kopie des Rumpfes der Funktion

angefertigt werden muß; in dieser Kopie werden dann die Werte der formalen Parameter ersetzt. Um dieses Problem zu umgehen arbeitet man üblicherweise mit Paaren, bestehend aus einem auszuwertenden Ausdruck e und einer Belegung für die freien Variablen von e.

Bei der Auswertung von Ausdrücken werden wir daher Ausdrücke nicht isoliert, sondern ihre Auswertung stets relativ zu einer *Umgebung* betrachten. Eine Umgebung ist dann eine Abbildung, die den freien Variablen eines Ausdrucks einen Wert im Speicher, also einen Knoten im Graphen zuordnet. Wir schreiben $u(x)$ für den Knoten, den die Umgebung u der Variable x zuordnet.

Unter dem Begriff *Abschluß* (englisch *closure*) verstehen wir ein Paar der Form $\langle u, e\rangle$ mit einer Umgebung u und einem Ausdruck e, wobei u allen freien Bezeichnern von e einen Knoten zuordnet.

2.8 Auswertung von Ausdrücken

Wir beschreiben im folgenden die Auswertung der verschiedenen Ausdrücke von L– aus Abbildung 2.1. Die Auswertung eines Ausdrucks e erfolgt relativ zu einer Umgebung u. Die Auswertung erfolgt schrittweise, wobei jeweils zu einer neuen Umgebung u' und einem neuen Ausdruck e' übergegangen wird:

$$\langle u, e\rangle \to \langle u', e'\rangle \to \ldots$$

Die Auswertung terminiert, wenn $\langle u, e\rangle$ nicht weiter reduzierbar ist. In diesem Fall spezifiziert $\langle u, e\rangle$ dann einen Knoten im Speicher. Wir identifizieren daher im folgenden das Paar $\langle u, e\rangle$ bei Terminierung der Auswertung mit einem Knoten im Graphen.

Unter einem *Programm* verstehen wir einen Ausdruck e, der keine freien Variablen enthält. Die Auswertung eines Programmes startet mit einer leeren Umgebung u.

Variable

Die Auswertung von $\langle u, x\rangle$ terminiert und liefert als Wert $u(x)$.

Zahlen

Die Auswertung von $\langle u, n\rangle$ terminiert durch Erzeugung eines neuen Zahl-Knotens mit Wert n.

Funktions-Definition

SML erlaubt nur die Definition einstelliger Funktionen. Mehrstellige Funktionen können entweder durch Currying oder die Übergabe von Tupeln dargestellt werden. Im Gegensatz zu SML erlauben wir in L direkt die Definition mehrstelliger Funktionen.

Die Auswertung einer Funktions-Definition

$$\langle u, \texttt{fn}\ (\overline{x})\ \texttt{=>}\ e\rangle$$

terminiert und liefert als Wert einen neuen Funktions-Knoten v, der mit `FUN`/**`fn`** $(\overline{x})$ **`=>`** e markiert ist. Von v führen Kanten weg, die auf die freien Variablen von e verweisen und die gemäß der aktuellen Umgebung u bestimmt werden.

Applikation

Bei der Evaluierung einer Applikation

$$\langle u, e_0(e_1, \ldots, e_n)\rangle$$

muß beachtet werden, daß die Applikation überladen ist: je nachdem ob e_0 zu einem Konstruktor oder einer Funktion reduziert, wird entweder ein neues Tupel erzeugt oder eine Funktion aufgerufen. Genauer: die Evaluierung erfolgt strikt, das heißt zunächst werden alle e_i von links nach rechts ausgewertet, was $v_0, \ldots, v_n$ liefert. Die Reihenfolge ist wichtig wegen Seiteneffekten z.B. bei Zellen. Dann wird eine Fallunterscheidung über v_0 vorgenommen. Handelt es sich bei v_0 um einen Konstruktor-Knoten, so wird ein neues Tupel mit Marke v_0 und Komponenten $v_1 \ldots v_n$ im Speicher erzeugt.

Handelt es sich bei v_0 um einen Funktions-Knoten, der mit `FUN`/**`fn`** $(\overline{x})$ **`=>`** e markiert ist, dann führt die Auswertung der Applikation zu einem neuen Abschluß $\langle u', e\rangle$. u' geht dabei aus u hervor, indem u um die Werte der freien Variablen von e erweitert wird, was ja gerade den von v_0 wegführenden Kanten entspricht. Zudem bildet u' die formalen Parameter x_i auf deren aktuelle Werte v_i ab.

Man beachte, daß in L eine Unterversorgung mit Parametern nicht erlaubt ist, da mehrstellige Funktionen ja nicht durch Currying dargestellt werden. Definiert man also etwa eine Funktion `f`

```
fun f(x,y) = x+y
```

und will man daraus eine einstellige Funktion erzeugen, die zu ihrem Argument die Zahl 4 addiert, so liefert

```
f(4)
```

einen Laufzeitfehler. Man muß stattdessen etwas umständlicher

```
fn(y)=>f(4,y)
```

schreiben. Dies entspricht auch der Situation in SML, wenn man `f` wie oben über ein zweistelliges Tupel als Argument definiert. Anders als in L kann man `f` aber in SML auch noch als

```
fun f x y = x+y
```

definieren und damit dann über

```
f 4
```

die gewünschte Funktion erhalten.

Konstruktor-Deklaration

Die Auswertung von

$$\langle u, \texttt{let con } x \texttt{ in } e \texttt{ end}\rangle$$

erzeugt einen neuen Konstruktor-Knoten v im Speicher und reduziert zu $\langle u', e\rangle$, wobei u' aus u hervorgeht, indem zusätzlich noch x auf v abgebildet wird.

Konstruktoren entsprechen in ihrer Verwendung weitestgehend den Konstruktoren aus SML. Sie können zur Konstruktion von Aufzählungstypen und zusammengesetzten Typen (in L in Form von Tupeln) verwendet werden. Sie unterscheiden sich aber von den SML-Konstruktoren darin, daß es sich dabei nicht um Funktionen sondern um symbolische Konstanten handelt. So kann etwa ein Konstruktor als Marke verschiedener Tupel mit unterschiedlicher Arität dienen.

Man beachte, daß die Konstruktor-Deklaration jeweils *neue* Knoten erzeugt. So führt also eine zweimalige Auswertung von

```
let con Hello in Hello end   ≠   let con Hello in Hello end
```

zu jeweils neuen verschiedenen Knoten im Speicher.

Variablen-Deklaration

Die Semantik von

$$\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}$$

ist äquivalent zu

$$(\texttt{fn } (x) \texttt{ => } e_2)(e_1)$$

mit einer neuen Variable x, die nicht frei in e_2 vorkommt. Somit ist diese Form der Deklaration redundant, allerdings würde eine direkte Umsetzung obiger Spezifikation jeweils zur Erzeugung eines neuen Funktions-Knotens führen. Wir definieren daher die Evaluierung von

$$\langle u, \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}\rangle$$

so, daß zunächst $\langle u, e_1\rangle$ evaluiert wird, was den Knoten v_1 liefert. Dann wird zu $\langle u', e_2\rangle$ reduziert, wobei u' aus u hervorgeht, indem zusätzlich noch x auf v_1 abgebildet wird.

Man beachte, daß (wie in SML) per Definition x in e_1 noch nicht sichtbar ist, dennoch wird ein rekursives **let** (wie etwa **let rec** in SML) in L nicht benötigt. Wie wir später sehen werden, kann das gewünschte Verhalten durch Verwendung einer logischen Variablen erreicht werden.

Rekursive Funktion

Da L– noch keine logischen Variablen kennt, erlauben wir die Definition rekursiver Funktionen mittels

$$\texttt{let rec } x = \texttt{fn } (\overline{y}) \texttt{ => } e_1 \texttt{ in } e_2 \texttt{ end}$$

wobei wie in SML hier x auch im Rumpf e_1 der Funktion bereits sichtbar ist. Damit können dann in L– Funktionen definiert werden, die sich selbst rekursiv aufrufen. Eine Definition wechselseitig rekursiver Funktionen ist in L– prinzipiell auch möglich aber syntaktisch aufwendiger. Wie wollen diesen Punkt aber hier nicht weiter vertiefen, da sich diese in L sehr viel einfacher durch die Verwendung logischer Variablen definieren lassen.

Case

Auch die Semantik des Case-Ausdrucks von L– entspricht (im Gegensatz zu L, wie wir später sehen werden) dem entsprechenden Konstrukt aus SML.

In einem Case-Ausdruck der Form

```
case e of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

bezeichnen wir die p_i als *Wächter* (engl. *guards*), die e_i als *Rümpfe*. Wie aus Abbildung 2.1 hervorgeht, handelt es sich bei einem Wächter um einen geschachtelten Ausdruck bestehend aus Tupeln, Konstanten und Variablen. Wie in SML verlangen wir, daß jede Variable x, die an Markenposition eines Tupels vorkommt, zuvor mittels

```
let con x in ... end
```

als Konstruktor eingeführt wurde.

Ein Wächter p_i dient als Variablen-Binder: genau wie in SML wird eine Variable x deklariert, wenn sie nicht über eine Konstruktor-Deklaration eingeführt wurde. Somit kommen hierfür höchstens solche Variablen in Frage, die in p_i nicht an Markenposition eines Tupels verwendet werden. Der Skopus von x erstreckt sich auf den zu p_i gehörenden Rumpf e_i.

Wie bereits erwähnt, sind in L– wie in SML (und im Gegensatz zu L) nur lineare Wächter erlaubt, das heißt, eine durch den Wächter deklarierte Variable darf im Wächter selbst nicht mehrfach vorkommen.

In folgendem Beispiel

```
case x of
  F(u,3,G(w,v)) => ...
| H(A(),S(y)) => ...
```

müssen also gerade `F`, `G`, `H`, `S` und `A` über Konstruktor-Deklarationen eingeführt worden sein. Dagegen werden `u`, `w`, `v` und `y` durch den Wächter deklariert, wenn sie nicht zuvor über eine Konstruktor-Deklarationen definiert wurden.

Die Auswertung des Case-Ausdruckes in L– ist genau wie in SML definiert. Zunächst wird e ausgewertet, was einen Knoten v liefert. Danach wird geprüft, ob ein Match von v mit einen der Wächter p_i erfolgreich ist; in diesem Fall reduziert der case-Ausdruck zu dem korrespondierenden e_i. Die Wächter werden dabei *sequentiell* geprüft: wenn zwei Wächter p_i und p_k mit

$i < k$ erfüllt sind, dann wird stets zu e_i reduziert. Ist kein Wächter erfüllt, so führt dies zu einem Laufzeitfehler (vgl. Abschnitt 2.12).

Beispielsweise prüft folgender Ausdruck, ob es sich bei `x` um eine Liste handelt:

```
case x of
  h::r => e1
| []   => e2
```

Ist `x` etwa an `[1,2,3]` gebunden, dann reduziert der Ausdruck zu e_1, bei dessen Auswertung dann `h` auf `1` und `r` auf `[2,3]` verweisen.

Eine explizite else-Klausel ist nicht nötig, hierzu dient ein Wächter, der nur aus einer neuen Variablen besteht:

```
case x of
  5 => e1
| y => e2
```

Ist in obigem Ausdruck `x` verschieden von `5`, so wird stets zu e_2 reduziert.

Man beachte, daß in L– ein Test, ob zwei Variablen x und y an den gleichen Wert gebunden sind, nicht möglich ist. Hierzu wird die erweiterte Definition des Case-Ausdrucks von L benötigt, worauf wir später zurückkommen werden.

2.9 Operationen in L–

Weitere Funktionalität wird durch die Verwendung von Operationen eingebracht. Dabei handelt es sich um vordefinierte Funktionen. Abbildung 2.2 zeigt die Operationen von L. L ist zwar nicht statisch getypt, wir haben aber dennoch in Abbildung 2.2 zur besseren Orientierung die Ein- und Ausgabetypen der einzelnen Operationen mit aufgenommen.

2.9.1 Arithmetik

Die Bedeutung der arithmetischen Operationen entspricht den üblichen Konventionen. Die Vergleichsoperation `<` liefert ein boolesches Ergebnis, was wir, wie in Abschnitt 2.2 beschrieben, durch die Konstruktoren **`true`** und **`false`** darstellen.

2.9.2 Zellen

Zellen in L entsprechen genau den Referenzen aus SML. So erzeugt eine Anwendung von

`ref(`e`)`

einen neuen `CELL`-Knoten im Speicher, dessen Inhalt der Wert von e ist. Allerdings ist `ref` in L eine Funktion und nicht wie in SML ein Konstruktor. Das hat zur Folge, daß `ref` in L nicht im Wächter eines Case verwendet werden darf, was wiederum entscheidend für die Einhaltung der Monotonie-Eigenschaft ist, auf die wir in Abschnitt 2.11.2 eingehen werden.

Anders als in SML umfaßt die Definition von L mit `exchange` lediglich eine einzige Operation zum Zugriff auf Zellen:

`exchange(`e_1`,`e_2`)`

erwartet, daß die Auswertung von e_1 einen Zell-Knoten liefert und schreibt dann in einem *atomaren* Schritt den Wert von e_2 in die Zelle und liefert als Wert den alten Inhalt der Zelle zurück. Die Möglichkeit, in einem atomaren Schritt den Inhalt einer Zelle zu lesen und gleichzeitig zu ändern, ist wesentlich für die nebenläufige Programmierung. Wir werden darauf in Abschnitt 2.13 zurückkommen; [VHB$^+$97] zeigt, wie man dadurch neben logischen Variablen noch komplexere Konstrukte (Locks) zur Synchronisation von Threads definieren kann.

Die `exchange`-Operation ist als Primitiv ausreichend, um die Funktionen `!` und `:=` zu realisieren, die nur lesend respektive nur schreibend auf eine Zelle zugreifen. Am einfachsten läßt sich `:=` definieren, indem man einfach `exchange` aufruft, dessen Ausgabe ignoriert und **`unit`** zurückliefert:

```
fun :=(cell,new) =
  exchange(cell,new);
  unit
```

Die Definition von `!` in L– zum lesenden Zugriff auf eine Zelle wird ein wenig aufwendiger:

```
fun !(cell) =
  let val old = exchange(cell,4711)
  in
      exchange(cell,old);
      old
  end
```

Man liest zunächst den alten Wert mittels `exchange` aus, merkt sich diesen in der Variablen `old` und schreibt einen beliebigen Wert (z.B. die Zahl 4711) in die Zelle. Durch einen weiteren Aufruf von `exchange` wird der ursprüngliche Inhalt der Zelle wiederhergestellt. Dieses Vorgehen funktioniert in L–, da L– nicht nebenläufig ist und daher die beiden Aufrufe von `exchange` nicht unterbrochen werden können. Im nebenläufigen Kontext von L verwendet man daher eine andere Möglichkeit der Implementierung der Funktion, die auf logische Variablen zurückgreift.

Wir schließen damit die Betrachtungen zu L– ab und wenden uns im folgenden der Erweiterung von L– zu L zu.

2.10 Logische Variablen

Logische Variablen erlauben die Darstellung von unvollständiger Information im Speicher, nämlich daß der Wert einer Variablen noch unbestimmt ist. Im Graphen werden logische Variablen als Knoten dargestellt, die mit `VAR` markiert sind. Gemäß Definition von $\sim$ aus Abschnitt 2.6 ist die Gleichheit von Variablen über die Gleichheit von Knoten definiert.

Der Begriff „Variable“ wird nun überladen verwendet. Zum einen bezeichnen wir damit die Variablen, die in Programmen vorkommen, und die auf Knoten im Speicher verweisen. Zum anderen bezeichnen wird damit die mit `VAR` markierten Knoten im Speicher. Wenn im folgenden daher aus dem Kontext nicht klar erkennbar ist, welche Bedeutung gemeint ist, sprechen wir von den ersteren auch als „Programmvariablen“, während wir die `VAR`-Knoten im Speicher als „logische Variablen“ bezeichnen.

2.10.1 Erzeugung

In L werden neue logische Variablen nie implizit sondern stets explizit durch Aufruf der nullstelligen Operation `lvar`() erzeugt, deren Auswertung einen neuen Knoten im Speicher erzeugt, der mit `VAR` markiert ist.

2.10.2 Unifikation

Unter *Unifikation* verstehen wir das Gleichmachen zweier Knoten K und K'. Dazu wird eine *minimale* Anzahl von `VAR`-Knoten eliminiert, indem man diese Knoten mit anderen Knoten verschmilzt, so daß dann $K \sim K'$ gilt. Das Verschmelzen eines `VAR`-Knotens mit einem anderen Knoten bezeichnen wir auch als *Binden* einer Variablen. Abbildung 2.5 verdeutlicht das Vorgehen bei der Unifikation zweier Tupel.

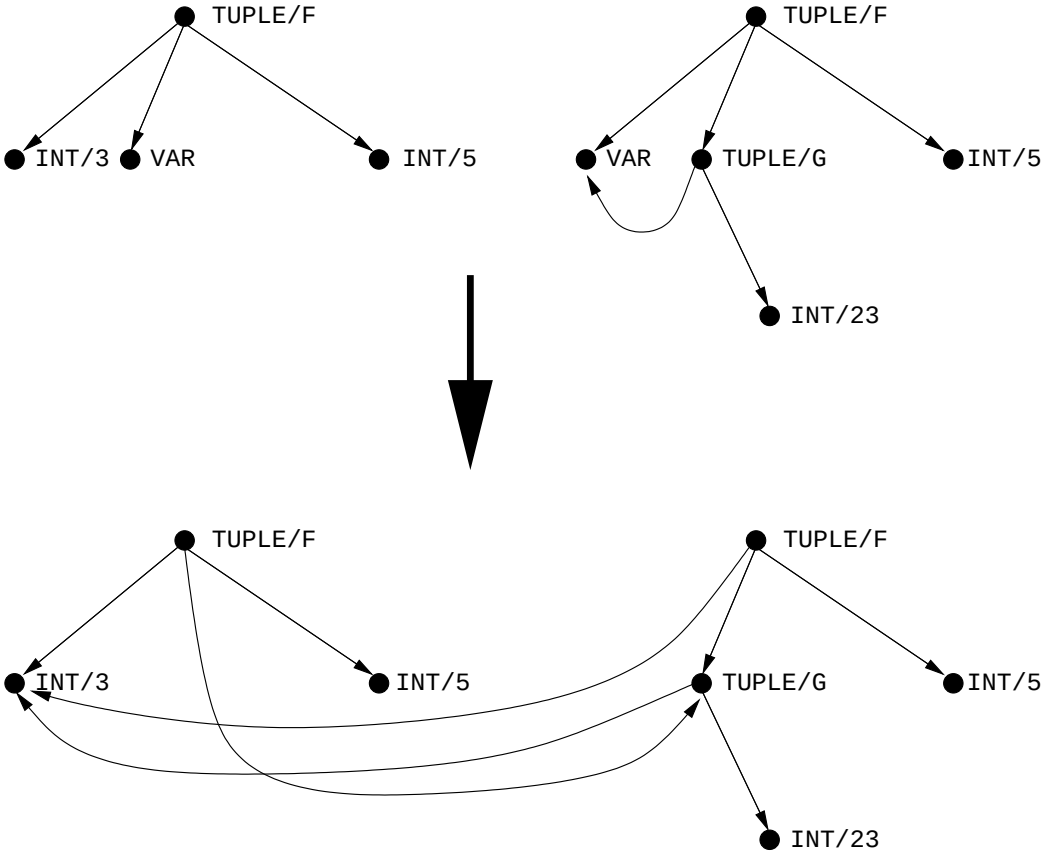
Wenn es keine Möglichkeit gibt, durch Binden von Variablen die Gleichheit herzustellen, dann sagen wir, daß die Unifikation fehlschlägt. Man beachte, daß bei erfolgreicher Unifikation wegen des Minimalitätskriteriums die Menge der zu eliminierenden Variablen eindeutig bestimmt ist.

Eine Unifikation geschieht stets explizit durch Anwendung der Operation `unify`(x,y), die eine Unifikation von x mit y im Speicher durchführt. Dabei wird davon ausgegangen, daß dies immer möglich ist. Schlägt die Unifikation fehl, so wird eine Fehlerbehandlung durchgeführt (vgl. Abschnitt 2.12). Die Auswertung von `unify` liefert daher stets **`unit`** als Ergebnis.

2.10.3 Zyklische Datenstrukturen

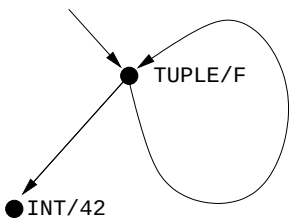
Durch die Verwendung von logischen Variablen und Unifikation wird es nun in L möglich, zyklische Datenstrukturen zu erzeugen. So erzeugt die Auswertung von

Abbildung 2.5 Unifikation zweier Tupel




```
let con F;
    val y = lvar();
    val x = F(42,y)
in
   unify(x,y);
   x
end
```

folgendes zyklisches Tupel im Speicher:



Aus Sicht des Constraintmodells handelt es sich dabei um einen rationalen Baum. Rationale Bäume sind möglicherweise unendliche Bäume, die durch zyklische Graphen endlich dargestellt werden können. Im Beispiel oben handelt es sich um den unendlichen Baum der Form `F(42,F(42,F(42,...)))`.

Eine andere Form von Zyklen sind Funktionen, die sich wechselseitig rekursiv aufrufen. Eine direkte Definition etwa in der Form

```
let val f = fn(x)=>g(x);
    val g = fn(x)=>f(x)
in ... end
```

ist wegen der Sichtbarkeitsregeln nicht möglich. Dennoch braucht L kein **let rec** wie SML, man kann das gewünschte Ergebnis durch Erzeugung logischer Variablen mit anschließender Unifikation erreichen:

```
let val f = lvar();
    val g = lvar()
in
   unify(fn(x)=>g(x), f);
   unify(fn(x)=>f(x), g);
   ...
end
```

2.10.4 Suspension

Anders als in L– kann es in L durch die Existenz der logischen Variablen passieren, daß beispielsweise eine Addition `x+5` nicht ausgewertet werden kann, weil `x` auf einen `VAR`-Knoten verweist. Dies führt nun aber nicht zu einem Laufzeitfehler, vielmehr wird die Auswertung der Addition einfach zurückgestellt oder *suspendiert* und mit der nebenläufigen Auswertung eines anderen Ausdrucks fortgefahren (mehr dazu in Abschnitt 2.11). Die Auswertung von `x+5` wird dann wieder aufgenommen, sobald `x` durch eine Unifikation gebunden wird.

Es gibt in L somit Ausdrücke, die immer auswertbar sind, während manche Ausdrücke nur ausgewertet werden können, wenn der Inhalt des Speichers bestimmte Bedingungen erfüllt. Wir sprechen dann davon, daß die Evaluierung dieser Ausdrücke auf den Inhalt des Speichers *synchronisiert*. Dabei wird immer auf *Determiniertheit* synchronisiert, das heißt, daß bestimmte Knoten im Speicher nicht mit `VAR` markiert sind.

Wir beschreiben im folgenden die Synchronisationsbedingungen derjenigen Ausdrücke und Operationen von L, deren Auswertung suspendieren kann. Alle anderen Konstrukte können nicht suspendieren.

- Die arithmetischen Operationen synchronisieren darauf, daß ihre beiden Eingabeargumente determiniert sind.
- Die `exchange` Operation synchronisiert auf Determiniertheit des ersten Argumentes (der Zelle), beim zweiten Argument darf es sich um eine logische Variable handeln.
- Die `spawn` Operation (siehe Abschnitt 2.11) synchronisiert auf Determiniertheit ihres Eingabeargumentes.
- Die Applikation $e_0(e_1, \ldots, e_n)$ suspendiert, wenn die Auswertung von e_0 eine logische Variable liefert. (Man beachte den Unterschied zwischen Suspension bei der Auswertung von e_0 einerseits und einer terminierenden Auswertung von e_0, die als Ergebnis eine Variable liefert.)

- Die Auswertung des Case-Ausdrucks kann in L suspendieren, wenn kein Wächter erfüllt ist. Wir werden im folgenden Abschnitt ausführlich darauf eingehen.

2.10.5 Case

In L erlauben wir, wie bereits erwähnt, in einem Case-Ausdruck der Form

```
case x of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

auch nicht-lineare Wächter p_i, das heißt, Variablen, die von einem Wächter eingeführt werden, dürfen in diesem auch mehrfach vorkommen. Da zudem L im Gegensatz zu L– auch logische Variablen enthält, müssen wir die Semantik anders definieren als in L–, indem wir die Begriffe *Subsumption* (engl. entailment) und *Dissubsumption* (engl. disentailment) einführen.

Zum besseren Verständnis beleuchten wir kurz die Bedeutung der Begriffe im Constraintmodell: im Constraintmodell faßt man den Speicher als prädikatenlogische Formel C auf. x aus dem Case-Ausdruck oben wird dann von einen Wächter p_i subsumiert, wenn $\exists\overline{y}(x = p_i)$ logisch von C impliziert wird, wobei $\overline{y}$ die vom Wächter neu deklarierten Variablen sind, alle anderen Variablen sind implizit allquantifiziert. Wenn die Negation dieser Gleichung von C impliziert wird, ist der Wächter dissubsumiert. Eine dritte Möglichkeit ist die, daß keine der beiden Bedingungen erfüllt ist, weil bestimmte logische Variablen noch nicht determiniert sind.

Ein Case-Ausdruck in L wird wie in L– sequentiell ausgewertet, indem für jeden Wächter sukzessive Subsumption geprüft wird. Liegt Dissubsumption vor, so wird der nächste Wächter getestet. Kann weder Sub- noch Dissubsumption entschieden werden, so suspendiert die Auswertung des Ausdrucks. Es wird also nicht mit der Prüfung des folgenden Wächters fortgefahren, sondern darauf gewartet, daß mehr Information in den Speicher geschrieben wird, so daß die Auswertung an dieser Stelle wieder aufgenommen werden kann.

In unserem operationalen Modell prüfen wir Subsumption und Dissubsumption eines Wächters p_i wie folgt (zum formalen Hintergrund verweisen wir auf [ST94]): bei jeder Auswertung des Case-Ausdrucks bauen wir gemäß p_i einen Knoten v_i im Speicher auf,[4] wobei wir für die neu deklarierten Variablen $\overline{y}$ `VAR`-Knoten erzeugen. v_i kann also nur Tupel, Variablen, Konstruktoren und Zahlen enthalten. Danach wird versucht, x (der Knoten, über den die Fallunterscheidung durchgeführt wird) und v_i zu unifizieren. Schlägt diese Unifikation fehl, dann liegt Dissubsumption vor. Ist die Unifikation erfolgreich, muß noch entschieden werden, ob Subsumption vorliegt oder ob suspendiert werden muß. Subsumption liegt anschaulich gesprochen genau dann vor, wenn bei der Unifikation von x mit v_i keine Änderungen an x (und dem darüber erreichbaren Teilgraphen) vorgenommen wurden. Das heißt, daß bei der Unifikation höchstens die Variablen aus $\overline{y}$ gebunden wurden, was wiederum gerade der Realisierung des Existenzquantors entspricht.

Wird x von einem Wächter p_i mit neu deklarierten Variablen $y_1, \ldots, y_n$ subsumiert, so reduziert die Auswertung des obigen Case-Ausdruckes zu

[4] In Abschnitt 3.9 werden wir sehen, daß die Kunst der Implementierung darin besteht, ein explizites Erzeugen von neuen Tupeln im Speicher so weit wie möglich zu umgehen.

```
let val y1 = lvar();
    val y2 = lvar();
    ...
    val yn = lvar()
in
    unify(x,pi);
    ei
end
```

Beispiele

Wir verdeutlichen die Verwendung des Case-Konstrukts an verschiedenen Beispielen.

Betrachten wir zunächst die Funktion `append`, die zwei Listen konkateniert:

```
fun append(x,y) =
  case x of
    h::r => h::append(r,y)
  | []   => y
```

Man beachte, daß wie oben beschrieben, das Konstrukt als Variablen-Binder für `h` und `r` dient, die bei der Unifikation im Wächter an das erste Element der Liste `x`, respektive den Schwanz von `x` gebunden werden.

Oft will man die Auswertung eines Ausdruckes so lange suspendieren, bis eine Variable `x` determiniert ist, das heißt bis `x` auf einen beliebigen Knoten im Speicher verweist, der nicht mit `VAR` markiert ist. Dieses Verhalten realisiert die Funktion `wait`, die sich wie folgt formulieren läßt:

```
fun wait(x) =
  case x of
    1 => unit
  | y => unit
```

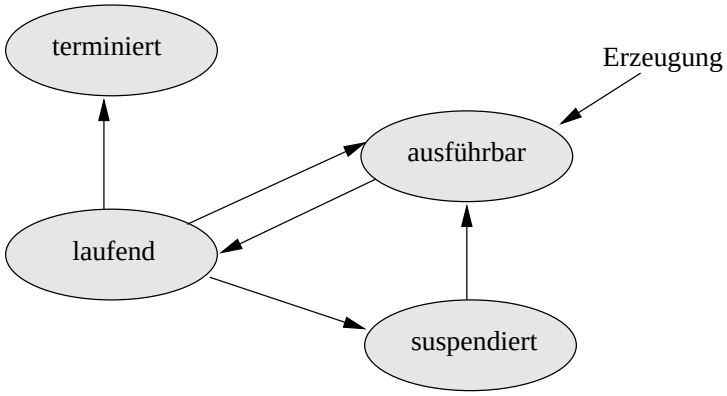
Erst wenn entschieden werden kann, ob `x` gleich oder verschieden von der Zahl 1 ist, kann das case reduzieren, was wiederum genau dann der Fall ist, wenn `x` determiniert ist.

Diese Beispiele nutzen nur die eingeschränkten Möglichkeiten von **`case`**, die auch in SML verfügbar sind. Man kann damit aber auch noch mehr anfangen: so kann man etwa mittels

```
case x of
  u::v::u::v::w => e1
| y             => e2
```

prüfen, ob das erste und dritte und zugleich das zweite und vierte Element einer Liste gleich sind, was in SML nicht so einfach zu formulieren wäre. Auch wenn nur partielle Information vorhanden ist, kann der Ausdruck unter Umständen bereits reduziert werden. Das ist etwa der Fall, wenn das zweite Element der Liste die Zahl 4 und das vierte die Zahl 10 sind, alle anderen Elemente (insbesondere vielleicht auch die Länge der Liste) noch unbestimmt (d.h. `VAR`-Knoten) sind. Dies verdeutlicht die Verbindung zum Constraint Logischen Programmieren (CLP).

Abbildung 2.6 Zustände und Zustandsübergänge von Threads



Mehrere Tests können in einem Wächter durchgeführt werden, indem man diese beispielsweise in eine Liste einbettet. Will man etwa prüfen, ob die Variablen `x` den Wert 5 und `y` den Wert 11 haben, so geht dies über:

```
case [x,y] of
  [5,11] => e1
| z      => e2
```

Man beachte, daß bereits zu e_2 reduziert werden kann, wenn `x` noch auf einen `VAR`-Knoten, aber `y` beispielsweise schon auf die Zahl 42 verweist.

Auch ein Test auf Gleichheit zweier Variablen `x` und `y` ist möglich:

```
case [x,y] of
  [z,z] => e1
| v     => e2
```

Weder `x` noch `y` müssen unbedingt determiniert sein: wenn beide auf den selben `VAR`-Knoten verweisen, kann bereits zu e_1 reduziert werden. Interessant ist auch der Fall, daß `x` auf ein Tupel der Form `F(u)` und `y` auf `F(v)` mit zwei verschiedenen `VAR`-Knoten `u` und `v` verweist. In diesem Fall muß die Ausführung suspendieren.

2.11 Threads

In diesem Abschnitt wollen wir uns der zweiten Erweiterung von L gegenüber L– zuwenden: Nebenläufigkeit durch Threads.

In L ist jeder Abschluß $\langle u, e\rangle$ zur Auswertung eines Ausdruckes e eindeutig einem *Thread* zugeordnet, der die Auswertung dieses Abschlusses übernimmt. Wie Abbildung 2.6 verdeutlicht, kann ein Thread sich in verschiedenen Zuständen befinden: verschiedene Threads werden nebenläufig ausgeführt, d.h. mehrere Threads können zeitgleich ausgeführt werden, wobei allerdings jeder einzelne Reduktionsschritt eines Ausdrucks eine atomare Operation darstellt. Wird der Abschluß eines Threads ausgeführt, so befindet sich der Thread im Zustand *laufend*; in der in dieser Arbeit

diskutierten sequentiellen Implementierung, kann es stets nur höchstens einen laufenden Thread geben. Ein Thread ist *ausführbar*, wenn sein Abschluß zwar auswertbar ist, er sich allerdings nicht im Zustand laufend befindet, weil die Ausführungsressourcen erschöpft sind.

Wenn der Abschluß eines Threads vollständig evaluiert wurde oder wenn ein Laufzeitfehler auftritt (siehe Abschnitt 2.12), geht ein Thread in den Zustand *terminiert* über.

Muß die Auswertung des Ausdruck e eines Threads T suspendieren (vgl. Abschnitt 2.10.4) dann geht T so lange in den Zustand *suspendiert* über, bis der Speicher genug Information enthält, so daß die Evaluierung von e wieder aufgenommen werden kann.

2.11.1 Fairneß

Die Ausführungsstrategie garantiert *Fairneß*: jeder ausführbare Thread wird nach endlicher Zeit ausgeführt. Die Garantie der Fairneßeigenschaft ist keine Selbstverständlichkeit in nebenläufigen Sprachen: so läßt beispielsweise der Java Sprachreport [GJS97] diesen Aspekt offen; hier ist es möglich, daß ein unendlich laufender Thread die Ausführung anderer Threads mit gleicher Priorität verhindert. Ähnliches gilt für den POSIX Threads Standard: eine Implementierung dieses Standards hat die Möglichkeit unterschiedliche Scheduler zur Verfügung zu stellen. Die Existenz eines fairen Schedulers ist nicht zwingend vorgeschrieben [But97].

2.11.2 Monotonie

Im letzten Abschnitt haben wir festgelegt, daß zwei lauffähige Threads T_1 und T_2 fair bedient werden. Nehmen wir an, daß in einer sequentiellen Implementierung zunächst T_1 evaluiert wird. Was passiert nun, wenn T_1 den Speicher so modifiziert, daß T_2 nun nicht mehr ausführbar ist, sondern suspendiert werden muß? Dadurch verliert nicht nur die Garantie der Fairneß an Bedeutung, vielmehr wird auch ein deterministisches Verhalten von Programmen erschwert, da dieses stark vom Scheduler abhängt.

In L kann nun die oben beschriebene Situation gar nicht erst eintreten. In L wird der Speicher während der Berechnung stets *konsistent* erweitert: ist ein Ausdruck auswertbar, weil eine bestimmte Bedingung vom Speicher erfüllt wird, so bleibt diese Eigenschaft unabhängig von der Auswertung anderer Ausdrücke erhalten.[5] Diese wichtige Eigenschaft der Sprache bezeichnen wie als *Monotonie*-Eigenschaft.

Die Einhaltung der Monotonie ist in L aus folgendem Grund gesichert: In den vorangehenden Abschnitten haben wir gesehen, daß die einzigen destruktiven Änderungen (also Änderungen von bereits existierenden Marken oder Kanten) am Speicher entweder durch Binden einer Variablen im Rahmen einer Unifikation oder durch Änderung des Inhalts einer Zelle erfolgen können. Sonst können keine bereits existierenden Kanten und nur die Marken von `VAR`-Knoten durch Verschmelzung geändert werden. Wir haben gesehen, daß nur auf die Änderung eines `VAR`-Knotens suspendiert werden kann. Es kann dagegen nicht darauf synchronisiert werden, daß ein Knoten mit `VAR` markiert ist oder der *Inhalt* einer Zelle einen bestimmten Wert hat (letzteres ist in SML sehr wohl möglich, da dort im Gegensatz zu L `ref` auch innerhalb des Wächters eines case-Ausdruckes erlaubt ist).

[5] Aus der Sicht des Constraintmodells, wo man den Speicher als logische Formel C auffaßt, heißt das, daß jede Änderung am Speicher einen Übergang zu einer neuen Formel C' bewirkt, die C logisch implizieren muß.

2.11.3 Erzeugung

Die Erzeugung eines neuen Threads geschieht durch

```
spawn(f)
```

Die Operation erwartet, daß es sich bei f um eine nullstellige Funktion handelt. f wird dann in einem neu erzeugten Thread appliziert. Die Auswertung von `spawn(`f`)` liefert stets **`unit`**. Das heißt also insbesondere, daß das Ergebnis der Auswertung von f verloren geht. Daher haben wir in Abbildung 2.3 eine syntaktische Abkürzung **`thread`** e definiert, die einen beliebigen Ausdruck e in einem neuen Thread auswertet und das Ergebnis dieser Auswertung zurückliefert:

```
let val x = lvar()
in
  spawn(fn()=>unify(x,e));
  x
end
```

Hier wird zunächst eine neue logische Variable `x` erzeugt. Diese wird dann im Rumpf einer anonymen Funktion an den Wert der Auswertung von e gebunden.

2.12 Fehlerbehandlung

Bei der Evaluierung können verschiedene Arten von *Laufzeitfehlern* auftreten. Eine Möglichkeit ist beispielsweise die Division durch 0, eine andere sind Typfehler, wie etwa der Versuch zwei Konstruktoren miteinander zu multiplizieren. Ein Laufzeitfehler kann auch bei der Funktionsapplikation auftreten, wenn eine falsche Zahl von Parametern verwendet wird. Eine weitere Fehlermöglichkeit ist der Versuch den Speicher inkonsistent zu erweitern, also beispielsweise die Zahlen 5 und 11 miteinander zu unifizieren.

Zur Behandlung von Laufzeitfehlern existiert in Mozart das Konzept der *Ausnahmebehandlung*, mit dem es möglich wird, solche Fehler auf Sprachebene abzufangen und entsprechend zu behandeln. Wir gehen im Rahmen dieser Arbeit nicht näher auf dieses Konzept ein und verweisen auf [Meh99]. Wir gehen stattdessen im folgenden davon aus, daß derjenige Thread, der einen Ausdruck evaluiert, der zu einem Laufzeitfehler führt, terminiert wird.

2.13 Beispiele

Wir werden im folgenden die Verwendung logischer Variablen, Nebenläufigkeit und Konstruktoren in L anhand einiger Beispiele verdeutlichen.

2.13.1 Top-down Konstruktion von funktionalen Datenstrukturen

Wir beginnen mit einem einfachen Beispiel, der Konkatenation von Listen, das bereits eine Facette der Ausdruckskraft der logischen Variable aufzeigt. Wir haben bereits gesehen, daß man die Funktion `append`, die zwei Listen `x` und `y` aneinander hängt, wie folgt formulieren kann:

```
fun append(x,y) =
  case x of
    h::r => h::append(r,y)
  | []   => y
```

Der Nachteil dieser Definition liegt darin, daß `append` nicht endrekursiv ist, das heißt der rekursive Aufruf von `append` im Rumpf ist nicht der letzte Ausdruck im Rumpf der Funktion, da danach noch die Konstruktion der Ausgabeliste erfolgen muß. Im Implementierungsteil werden wir sehen, daß endrekursive Funktionen sehr viel effizienter implementiert werden können. Unter Zuhilfenahme der logischen Variable läßt sich allerdings auch eine endrekursive Version formulieren. Wir definieren die Funktion `appendrec` als dreistellige Funktion, die stets **`unit`** zurückliefert und die ihr eigentliches Ergebnis im dritten Argument liefert, wo sie eine logische Variable erwartet, die an das Ergebnis der Konkatenation gebunden wird:

```
fun appendrec(x,y,res) =
  case x of
    h::r => let val aux = lvar()
            in  unify(res,h::aux);
                appendrec(r,y,aux)
            end
  | []   => unify(res,y);

fun append(x,y) =
  let val res = lvar()
  in  appendrec(x,y,res);
      res
  end
```

Durch die Erzeugung der logischen Variable `aux` kann so die Listenzelle bereits vor dem (nun end-)rekursiven Aufruf von `appendrec` erzeugt werden. Im Implementierungsteil und im Evaluierungsteil dieser Arbeit werden wir dann sehen, daß die zweite Formulierung der Funktion keinen Mehrverbrauch an Speicher mit sich bringt und zudem sehr viel effizienter ist, so daß sich in L (und damit auch in Oz) die Listenkonkatenation sogar deutlich schneller als in anderen funktionalen Sprachen, wie etwa in SML realisieren läßt. In funktionalen Sprachen ist eine direkte endrekursive Formulierung von `append` und vieler ähnlicher Funktionen nicht so ohne weiteres möglich; hier kann man allenfalls kompliziert umformulieren (z.B. unter Verwendung von Zellen, dann aber mit veränderter Signatur) oder das Problem seitens der Sprachimplementierung unter Verwendung elaborierter Implementierungstechniken angehen [BD99].

2.13.2 Sichere abstrakte Datentypen

Die sichere Implementierung abstrakter Datentypen bildet ein wichtiges Einsatzfeld von Konstruktoren. Wir wollen dies am Beispiel zweier Funktionen zur Codierung und Decodierung beliebiger Werte demonstrieren (vgl. hierzu auch [Smo97]).

Eine erste Möglichkeit zur Implementierung könnte dann wie folgt aussehen:

```
let con Key;
    fun encode(value) =
      Key(value);
    fun decode(secret) =
      case secret of
        Key(value) => value
in
   [encode,decode]
end
```

Der Ausdruck liefert als Ergebnis in einer Liste die Funktionen `encode` und `decode` zurück. Die Funktion `encode` nimmt einen beliebigen Wert und schließt diesen in ein Tupel mit Marke `Key` ein. Da der Skopus von `Key` auf die beiden Funktionen `encode` und `decode` beschränkt wird und zudem kein direkter Zugriff auf einzelne Argumente von Tupeln möglich ist, kann nun nur noch mit der Funktion `decode` der ursprüngliche Wert wieder ermittelt werden (wie in Abschnitt 2.8 beschrieben, führt ein Aufruf von `decode` mit einem unpassenden Wert für `secret` zu einem Laufzeitfehler). Die Funktion `encode` kann man nun weitergeben, so daß andere Daten verschlüsseln können, während `decode` geheim gehalten wird.

Leider ist dieses Vorgehen nun noch nicht völlig sicher: so hat ein Angreifer, der die Funktion `encode` kennt, die Möglichkeit die verschlüsselten Daten zu raten: hat er beispielsweise einen verschlüsselten Wert x in der Hand, so kann er herausfinden, ob es sich dabei um die verschlüsselte Zahl `5` handelt, indem er einfach `encode(5)` mit x vergleicht. Dieses Vorgehen funktioniert, da in L die Gleichheit von Tupeln über strukturelle Gleichheit definiert ist.

Wir können dieses Manko aber beseitigen, indem wir ausnutzen, daß Konstruktoren anders als etwa in SML auch dynamisch zur Laufzeit neu erzeugt werden können. Dazu erweitern wir `encode` so, daß bei jeder Verschlüsselung zusätzlich zu dem globalen Schlüssel `Key` noch jeweils ein eindeutiger Schlüssel `Id` erzeugt wird:

```
let con Key;
    fun encode(value) =
      let con Id in Key(Id,value) end;
    fun decode(secret) =
      case secret of
        Key(_,value) => value
in
   [encode,decode]
end
```

Dadurch führt ein mehrmaliges Verschlüsseln des gleichen Wertes zu jeweils verschiedenen Tupeln, so daß der Trick von oben nicht mehr anwendbar ist.

2.13.3 Synchronisierte Schlange

Wir verdeutlichen verschiedene weitere Aspekte der Sprache anhand eines komplexeren Beispiels, der Implementierung des abstrakten Datentyps einer *Schlange* (vgl. Abbildung 2.7). Es werden drei Funktionen zur Verfügung gestellt, die der Ausdruck in Abbildung 2.7 in Form einer

Liste `[newqueue,enqueue,dequeue]` zurückliefert: die nullstellige Funktion `newqueue` erzeugt ein neues Objekt des abstrakten Datentyps. `dequeue(q)` liefert das erste Element der Schlange `q` und entfernt dieses. `enqueue(q,x)` schließlich fügt ein neues Element `x` am Ende der Schlange `q` an.

Wir wollen die Implementierung der Schlange im folgenden genauer betrachten.

Abbildung 2.7 Implementierung einer Schlange in L

```
let con Key;
    fun newqueue() =
      let val c     = lvar();
          val first = ref(c);
          val last  = ref(c)
       in  Key(first,last)
       end;
    fun enqueue(q,x) =
      case q of
        Key(first,last) =>
          let val new = lvar();
              val old = exchange(last,new)
          in unify(x::new,old)
          end
    fun dequeue(q) =
      case q of
        Key(first,last) =>
          let val new = lvar();
              val x   = lvar();
              val old = exchange(first,new)
          in wait(old);
             unify(x::new,old);
             x
          end
in
   [newqueue,enqueue,dequeue]
end
```

Logische Variablen

Zur Implementierung der Schlange selbst verwenden wir eine Liste. Deren Ende bildet allerdings nicht wie sonst üblich der Konstruktor `nil`, sondern eine logische Variable `x`. Dadurch wird das Einfügen eines neuen Elementes `e` am Ende der Schlange sehr einfach: man muß lediglich eine neue logische Variable `y` erzeugen, und `x` mit `e::y` unifizieren.

Zellen

Zum Einreihen eines neuen Elementes, bzw. zum Ausreihen werden zwei Zellen `first` und `last` verwandt, die auf den Anfang respektive auf das Ende der Schlange verweisen. Verweist der Inhalt von `last` beispielsweise auf die Variable `x`, so wird der Inhalt von `last` nach dem Einfügen, wie im vorangehenden Abschnitt beschrieben, auf eine neue Variable `y` zeigen.

Man beachte, daß die Inhalte von `first` und `last` jeweils nicht direkt auf einzelne Elemente der Schlange verweisen, sondern auf entsprechende Sublisten. Auch ist zu beachten, daß bei einer leeren Schlange die Inhalte beider Zellen auf dieselbe Variable zeigen.

Private Konstruktoren

Der direkte Zugriff auf die Implementierung der Schlange (also auf `first` und `last`) soll lediglich den die Schlange implementierenden Funktionen `newqueue`, `enqueue` und `dequeue` vorbehalten bleiben. Hierzu betten wir `first` und `last` analog zum Vorgehen in Abschnit 2.13.2 in ein Tupel mit Marke `Key` ein, dessen Skopus wir auf die die Schlange implementierenden Funktionen beschränken.

Man beachte, daß dieser Konstruktor nur ein einziges Mal erzeugt wird und somit alle Schlangen den gleichen Konstruktor verwenden können. Es muß also nicht etwa bei der Erzeugung einer individuellen Schlange mittels `newqueue` jeweils ein neuer Konstruktor erzeugt werden.

Nebenläufigkeit

Die Implementierung der Schlange wurde so entworfen, daß sie auch gerade im nebenläufigen Kontext korrekt funktioniert. Zum einen arbeitet die Implementierung auch dann korrekt, wenn mehrere verschiedene Threads gleichzeitig auf dieselbe Schlange zugreifen. Zum anderen ist es wegen der Nebenläufigkeit der Sprache auch möglich, dem Ausfügen aus der Schlange auch dann ein sinnvolles Verhalten zuzuordenen, wenn die Schlange leer ist.

Wir wollen beide Fälle etwas genauer beleuchten und betrachten zunächst den konkreten Fall, daß zwei Threads T_1 und T_2 gleichzeitig mittels `enqueue` die Zahlen `1` respektive `2` in dieselbe Schlange `q` einfügen wollen (das Verhalten von `dequeue` ist in diesem Fall analog). Entscheidend für das korrekte Verhalten ist die Existenz der logischen Variable in L in Verbindung mit der Tatsache, daß die Operation `exchange` atomar ist.

Wir nehmen an, daß die logische Variable `x` das Ende von `q` darstellt, und daß zuerst T_1 `enqueue(q,1)` auswertet bis einschließlich der `exchange` Operation. Dadurch enthält dann die Zelle `last` eine neue Variable `x1`, während aber `x` noch nicht gebunden wurde. Wird nun die weitere Ausführung von `enqueue` in T_1 unterbrochen und zunächst `enqueue(q,2)` in T_2 vollständig ausgewertet, dann enthält `last` eine weitere neue Variable `x2` und `x1` ist an `2::x2` gebunden. Somit ist die Liste, die die Schlange darstellt zunächst unterbrochen; diese Unterbrechung wird aber beseitigt, sobald die Ausführung von T_1 wieder aufgenommen wird. Dann wird nämlich noch die Unifikation `unify(x,1::x1)` durchgeführt, wodurch die Liste wieder geschlossen wird.

Das Beispiel der Schlange zeigt noch einen weiteren interessanten Aspekt einer nebenläufigen Sprache in Verbindung mit logischen Variablen: so kann auch der Fall, daß ein Thread T_1 die Funktion `dequeue` aufruft, obwohl die Schlange leer ist, elegant behandelt werden, was in sequentiellen Sprachen zu einem Laufzeitfehler führen muß.

Eine Lösung kann so aussehen, daß T_1 einfach so lange suspendiert, bis ein anderer Thread T_2 die Funktion `enqueue` aufruft. Diese Möglichkeit haben wir in Abbildung 2.7 realisiert: der Aufruf von `wait(old)` innerhalb von `dequeue` bewirkt, daß der ausführende Thread so lange suspendiert wird, bis `old` gebunden ist. Nach Konstruktion kann `old` nur an eine Listenzelle gebunden werden, die wiederum nur von `enqueue` erzeugt werden kann.

Es gibt allerdings auch noch eine andere Möglichkeit, die einen noch höheren Grad an Nebenläufigkeit erlaubt. Hierzu muß lediglich der Aufruf `wait(old)` innerhalb von `dequeue` ersatzlos gestrichen werden: dann wird bei `dequeue(q)` und leerer Schlange wie zuvor einfach das Ende von `q` um eine neue Listenzelle verlängert, die eine neue logische Variable `x` enthält. `x` wird dann aber sofort ohne Suspension als Ergebnis zurückgeliefert; dadurch steht dann `last` noch *vor* `first`. Der Vorteil dieser Variante liegt darin, daß der ausführende Thread T_1 dann noch so lange weiterlaufen kann, bis zum ersten mal der Wert von `x` benötigt wird und erst dann suspendieren muß. Folgt später dann beispielswiese eine Einfügeoperation mittels `enqueue(q,23)` dann wird dadurch `x` an `23` gebunden und die Ausführung von T_1 kann wieder aufgenommen werden.

Zusammenfassung

- L ist eine *Teilsprache von Oz* mit einem *funktionalen Kern*. Zur Notation von L verwenden wir SML-Syntax.
- Wichtige Charakteristika von L sind: höhere Funktionen (mit call-by-value Semantik), logische Variablen, Threads, dynamische Typisierung.
- Variablen können als Werte Zahlen, Konstruktoren, Tupel, Zellen und Funktionen annehmen.
- *Logische Variablen* erlauben die explizite Darstellung von unvollständiger Information im Speicher.
- Der Case-Ausdruck ist ausdrucksstärker als in SML und erlaubt komplexe Synchronisationsbedingungen, da in L auch nicht-lineare Pattern erlaubt sind und die Semantik über logische Implikation definiert ist.
- Die Semantik wird informell über Auswertung von Abschlüssen über einem Graphen definiert.
- Ein *Abschluß* ist ein Paar $\langle u, e \rangle$ mit einer *Umgebung* u und einem *Ausdruck* e, wobei u allen freien Bezeichnern von e einen Knoten im Ausführungsgraphen zuordnet.
- *Threads* bestimmen die Reduktionsstrategie: jeder Thread wertet einen Ausdruck aus.
- Die Reduktionsstrategie garantiert *Fairneß*.
- Die *Monotonie*-Eigenschaft der Sprache garantiert, daß ein einmal reduzierbarer Thread immer reduzierbar bleibt.
- Logische Variablen bilden ein einfaches und transparentes Mittel zur *Synchronisation* von Threads.

Teil II

Maschinenmodell

Kapitel 3

Grundmodell

Das Graphenmodell aus dem vorangehenden Kapitel liefert zwar im Vergleich zum Constraintmodell schon wesentlich konkretere Anhaltspunkte für eine Implementierung von L, dennoch ist das Graphenmodell aus Implementierungssicht in verschiedener Hinsicht noch zu abstrakt. Daher wollen wir in diesem Kapitel ein maschinennäheres Modell vorstellen, das Modell einer *virtuellen Maschine*.

Sowohl zur Beschreibung der Implementierung von Programmiersprachen als auch zu ihrer Implementierung selbst greift man auf das Konzept einer virtuellen Maschine zurück. Diese Maschinen heißen „virtuell“, weil sie keine real existierende Hardware beschreiben (diese bezeichnen wir dann im folgenden zur Unterscheidung als *reale* Maschinen), sondern auf einem höheren plattformunabhängigen aber dennoch maschinennahen Abstraktionsniveau ansetzen. Programme für eine virtuelle Maschine können auf verschiedene Arten auf einer konkreten Hardwareplattform zur Ausführung gebracht werden: eine Möglichkeit – auf die wir uns in dieser Arbeit konzentrieren werden – besteht darin, den virtuellen Maschinencode von einem Interpretierer, auch *Emulator* genannt, ausführen zu lassen. Dieser wird aus Effizienzgründen in der Regel in einer hardwarenahen Sprache (C++ im Fall von Mozart) implementiert. Die Existenz einer virtuellen Maschine muß aber nicht unbedingt mit einem Emulator gekoppelt sein: so kann man in einem Übersetzungsschritt jede virtuelle Maschineninstruktion in eine Folge von realen Maschineninstruktionen der Zielplattform transformieren. Dieses Vorgehen hat den Nachteil, daß diese letzte Phase stark von der speziellen Zielplattform abhängig ist, und daher für jede unterstützte Plattform separat realisiert werden muß. Daher sind manche Systeme dazu übergegangen, den virtuellen Maschinencode in eine sowohl portable als auch hardwarenahe Zielsprache (wie zum Beispiel C) zu übersetzen [CFS94, Hau94, HCS95, GDBD92]. In beiden Fällen spricht man von einer *nativen* Ausführung des virtuellen Maschinencodes.

Die Wahl der Ausführungsmethode hat allerdings einen deutlichen Einfluß auf das Design der Instruktionen (weniger oder gar nicht auf die Daten- und Speicherstrukturen) der virtuellen Maschine. So versucht man beim emulatorbasierten Ansatz, den interpreterbedingten Mehraufwand dadurch zu minimieren, daß man den Instruktionssatz möglichst grobkörnig wählt, um so in einer Instruktion möglichst viel Arbeit zu erledigen. Auf der anderen Seite kann ein feinkörniger Befehlssatz beim nativen Ansatz viel besser den vom Compiler aquirierten Informationen Rechnung tragen. So kann man zum Beispiel überflüssige Typtests, Demaskierungen, Dereferenzierungen etc. einsparen. Wir werden in Abschnitt 6.7 noch genauer auf die Vor- und Nachteile von nativer Codeerzeugung im Gegensatz zu einem Emulator eingehen.

Bei der Darstellung von virtuellen Maschinen wird in der Literatur in der Regel direkt Bezug auf die Aspekte einer konkreten Implementierung genommen. Das erschwert meist den Einblick in die wesentlichen Ideen und Techniken, die dem Design zugrunde liegen, oder verschließt sogar dem in Implementierungsaspekten unerfahrenen Leser den Zugang gänzlich.

Wir wollen daher eine Beschreibung der virtuellen Maschine in verschiedenen Ebenen vornehmen. In diesem Teil der Arbeit geht es uns zunächst darum, ein Verständnis der Funktionsweise der Maschine auf möglichst hohem Abstraktionsniveau zu geben, bei dem allerdings bereits alle wichtigen Punkte Berücksichtigung finden. Wir werden uns daher zunächst noch nicht mit Bits, Bytes und Zeigern auseinandersetzen und auch völlig auf die detaillierte Präsentation von Algorithmen und Datenstrukturen zur Implementierung verzichten. Wir werden aber dann in Teil III genau auf die verschiedenen Aspekte einer konkreten Implementierung eingehen und dann dort auch genau sehen, wie sich das virtuelle Modell umsetzen läßt und welche Kompromisse unter Umständen auch aus pragmatischer Sicht heraus eingegangen werden müssen.

In den folgenden Abschnitten dieses Kapitels konzentrieren wir uns darauf, möglichst schnell ein kompaktes Gesamtbild zu vermitteln. Daher werden die in diesem Kapitel vorgestellten Techniken zwar immer korrekt sein, in manchen Fällen lassen sich aber noch Verbesserungen vornehmen. Wir wollen daher daran anschließend in einem eigenen Kapitel 4 ausführlicher auf verschiedene Optimierungsmöglichkeiten eingehen. Auch werden wir zugunsten einer kompakten Darstellung zunächst noch auf die Diskussion von Alternativen verzichten. Wir werden dann aber in Kapitel 5 auf mögliche alternative Realisierungen bei den unterschiedlichen Konstrukten genauer eingehen.

Wir werden in diesem Kapitel so vorgehen, daß wir zunächst nach einer kurzen Beschreibung des Zusammenspiels zwischen Maschine und Übersetzer auf die Änderungen eingehen, die die Darstellung des Graphen im Maschinenmodell erfährt. Anschließend stellen wir Register vor, die die Aufgabe der Umgebungen im Graphenmodell übernehmen. Darauf folgt eine Beschreibung des Formates von Maschinenprogrammen, dem sich eine ausführliche Diskussion von Suspension und Signalbehandlung anschließt. Darauf gehen wir dann auf die verschiedenen Instruktionen der Maschine ein. Wir schließen das Kapitel mit einem Ausblick auf die Aspekte von Persistenz und Verteilung ab.

Im folgenden gehen wir in Abschnitt 3.3 darauf ein, welche Änderungen sich am Graphen aus Sicht des Maschinenmodells ergeben. In Abschnitt 3.4 beschreiben wir die Adressierungsarten der Maschine. Abschnitt 3.7 geht auf die Implementierung von Suspension ein. In Abschnitt 3.8 folgt dann eine Beschreibung der Instruktionen der Maschine, wobei wir der Implementierung des Case-Ausdrucks einen eigenen Abschnitt widmen. Darauf zeigen wir in Abschnitt 3.10, daß eine Erweiterung der Maschine, die für inkrementelle und getrennte Übersetzung benötigt werden, sehr einfach realisierbar ist. Abschließend gehen wir in den Abschnitten 3.11 und 3.12 kurz auf zwei wichtige Aspekte neuerer Arbeiten an Oz ein, die Persistenz und vor allem Verteilung betreffen.

3.1 Eine neue Sprache

Das Modell, das wir in diesem Kapitel einführen, basiert auf einer Programmiersprache, die auf niedrigerem Abstraktionsniveau als L ansetzt. Wir bezeichnen sie daher als Maschinensprache, nennen ihre Anweisungen *Instruktionen* und bezeichnen ihr Ausführungsmodell als das Modell

einer *virtuellen Maschine*.[1] Die Maschineninstruktionen sind feinkörnig und von einem stark imperativen Charakter geprägt.

Da die Maschine nun nicht mehr direkt die Anweisungen von L versteht, muß in einem eigenen Übersetzungsschritt die Quellsprache von L in die Maschinensprache übersetzt werden, wobei die Baumstruktur der Ausdrücke von L linearisiert wird. Der *Übersetzer* transformiert L-Programme in äquivalente Sequenzen von Maschineninstruktionen. Die resultierenden Maschinenprogramme können wegen ihrer einfachereren Struktur und Granularität in der Regel sehr viel effizienter ausgeführt werden. Ein weiterer Vorteil ergibt sich dadurch, daß der Übersetzer nun bereits vor Programmausführung durch genaue Analyse des Quellprogramms viele Optimierungen durchführen kann, die in effizienterem Code resultieren [WM97].

Eine wichtige Aufgabe des Übersetzers besteht nun auch darin, die Korrektheit der erzeugten Programme zu gewährleisten. Die Maschinensprache ist nicht mehr typsicher, so daß es möglich wird, Maschinenprogramme mit undefiniertem Verhalten zu schreiben: wie wir weiter unten sehen werden, ist es beispielsweise möglich, eine Instruktion zu verwenden, die auf ein nicht existierendes Argument eines Tupels zugreift und so den Speicher korrumpieren kann. Auch ist es möglich, daß Maschinenprogramme die Fairneßeigenschaft verletzen: wie wir in Abschnitt 3.6 sehen werden, wird nur bei bestimmten Instruktionen auf Preemption getestet. Somit muß auch hier der Compiler sicherstellen, daß die Fairneßeigenschaft erhalten bleibt.

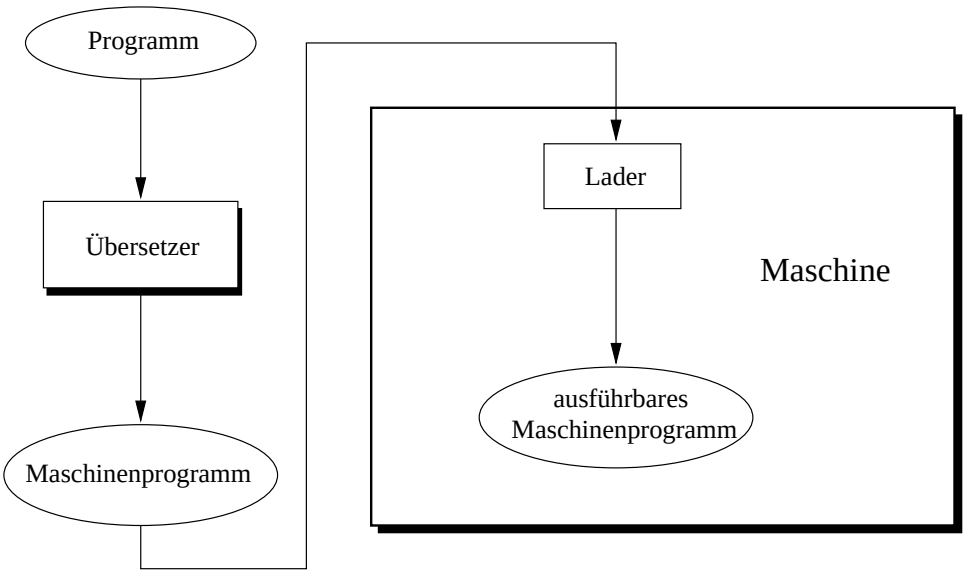
Eine interessante noch offene Frage ist die, ob es möglich ist, ähnlich wie für Java [LY96] zu einem gegebenen Maschinenprogramm zu entscheiden, ob dies ohne Verletzung von Konsistenz und Fairneß korrekt ausgeführt werden kann. Diese Fragestellung ist unter dem Aspekt der Verteilung [HVS97] (vgl. Abschnitt 3.12) von großer Bedeutung, da hier Maschinenprogramme über unsichere Medien verschickt werden.

3.2 Zusammenspiel Übersetzer und Maschine

Abbildung 3.1 zeigt ein vereinfachtes Modell, das das Zusammenspiel zwischen Übersetzer und Maschine verdeutlicht. Die Vereinfachung liegt darin, daß hier zunächst die Aspekte der getrennten Übersetzung und der Inkrementalität noch außer Acht gelassen wurden. Wir werden allerdings dann in Abschnitt 3.10 sehen, daß dieses einfache Modell sich sehr leicht entsprechend erweitern läßt. Der Übersetzer erhält als Eingabe ein *Programm*, also einen beliebigen L Ausdruck, der keine freien Variablen enthalten darf. Dieses Programm wird vom Übersetzer in ein *Maschinenprogramm* übersetzt und auf einer Datei abgelegt. In dieser Form kann das Maschinenprogramm nun noch nicht direkt von der Maschine ausgeführt werden. Hierzu ist ein spezieller Ladeschritt nötig: beim Hochfahren der Maschine transformiert daher der *Lader* die externe textuelle Darstellung des Maschinenprogramms in eine adäquate interne Repräsentation, so daß es in einem dafür erzeugten Thread ausgeführt werden kann. Die Maschine terminiert, sobald kein ausführbarer Thread mehr existiert.

[1] Genau genommen könnte man auch eine Implementierung des Graphenmodells als virtuelle Maschine bezeichnen, allerdings würde man hier wohl noch eher von einem Interpretierer sprechen.

Abbildung 3.1 Ein einfaches Modell zum Zusammenspiel zwischen Übersetzer und Maschine



3.3 Der Graph im Maschinenmodell

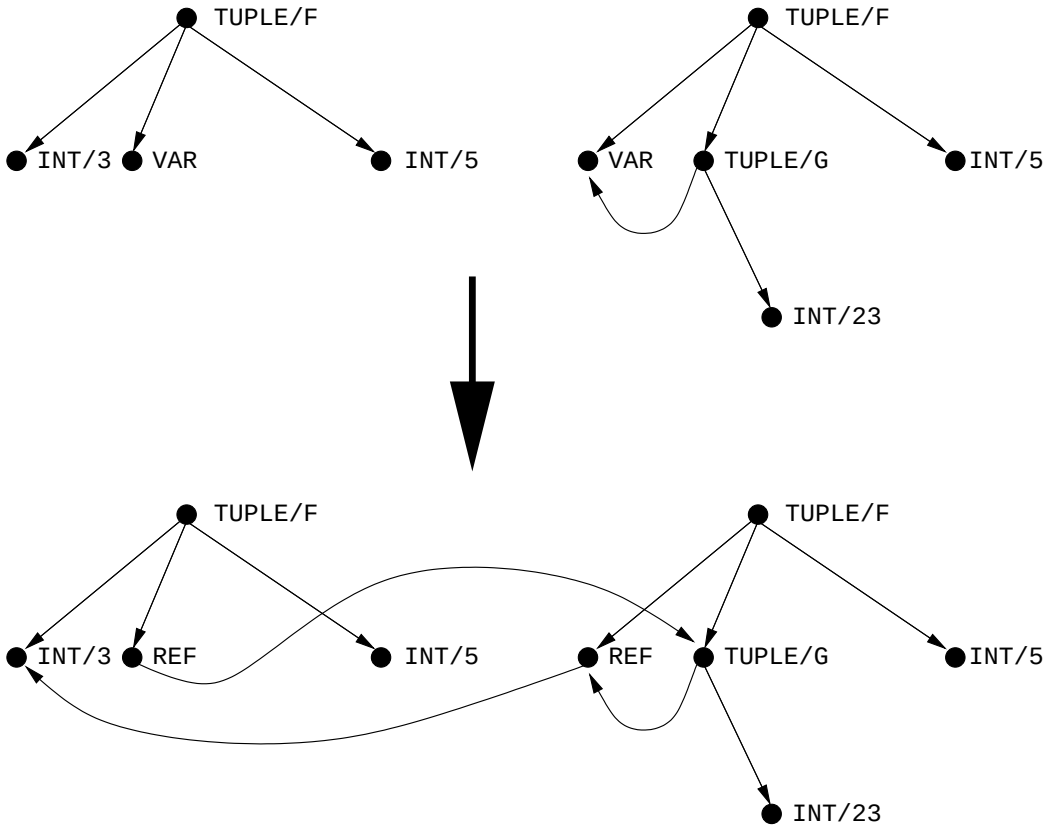
In diesem Abschnitt gehen wird auf die Änderungen bei der Darstellung des Graphen im Maschinenmodell gegenüber dem Graphenmodell aus Kapitel 2 ein. Da wir auch im Maschinenmodell noch soweit als möglich von Details der Implementierung abstrahieren wollen, werden wir das Modell aus Kapitel 2 direkt übernehmen und wenige Änderungen vornehmen, die wir im folgenden beschreiben. Auf die genaue Darstellung der Datenstrukturen im Speicher werden wir im Implementierungsteil III detailliert eingehen.

Die wesentliche Änderung im Speichermodell der Maschine betrifft die Darstellung der Variable: hier handelt es sich zwar nach wir vor um Knoten, die mit `VAR` markiert sind. Diese Knoten erhalten aber noch zusätzlich eine Referenz auf die sogenannte Suspensionsliste, auf die wir in Abschnitt 3.7.1 näher eingehen werden. Wichtiger aber ist eine Änderung, die sich im Zusammenhang mit der Unifikation ergibt: im Graphenmodell werden dabei `VAR`-Knoten mit anderen Knoten *verschmolzen*. Im Maschinenmodell gehen wir aus zwei Gründen anders vor: zum einen müßten beim Verschmelzen alle Kanten, die auf einen Knoten zeigen, geändert werden; es ist allerdings nur mit großem Aufwand (entweder an Speicherplatz oder Laufzeit) möglich, diese Kanten zu bestimmen. Zum anderen werden wir in Abschnitt 3.9 sehen, daß es zur Implementierung des Case nötig sein wird, Unifikationen auch wieder zurückzunehmen.

Anstatt einen `VAR`-Knoten K mit einem anderen Knoten K' zu verschmelzen, gehen wir nun so vor, daß wir die Marke von K in `REF` (für *Referenz*-Knoten) ändern und eine neue Kante einfügen, die von K zu K' führt. Abbildung 3.2 verdeutlicht, wie die Unifikation zweier Tupel, die wir in Abbildung 2.5 auf Seite 26 beim Graphenmodell verwandt hatten, nun unter Verwendung von `REF`-Zellen aussieht.

Die Maschine muß nun aber dennoch die Illusion erhalten, als wären Knoten verschmolzen worden. Daher behandelt die Maschine `REF`-Knoten transparent, das heißt wann immer der Wert eines Knotens benötigt wird, wird einer möglichen Referenzkette so lange gefolgt, bis ein Knoten er-

Abbildung 3.2 Unifikation ändert VAR-Knoten in REF-Knoten.

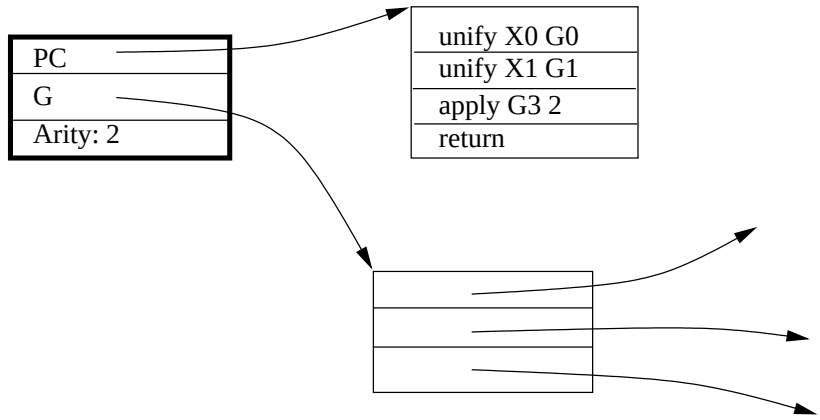


reicht wird, der nicht mit REF markiert ist. Diese Operation bezeichnen wir als *Dereferenzieren*. Das bedeutet nun allerdings nicht, daß die Maschine jedesmal, wenn eine Operation auf einem Knoten ausgeführt wird, zuvor einen Dereferenzierungsschritt durchführen muß. Dies ist in der Regel nur dann nötig, wenn der Wert des Knotens benötigt wird; zudem ist bei vielen Operationen bereits sichergestellt, daß ein Knoten keinen Referenz-Knoten enthalten kann. Wir werden diesen Aspekt noch ausführlich in Abschnitt 5.3 diskutieren und wollen hier daher nicht weiter darauf eingehen.

Die einzige wesentliche weitere Änderung bei der Darstellung von Werten im Speicher betrifft die Funktions-Knoten. Ein Funktions-Knoten wird nun markiert mit FUN/n, wobei n die Arität der Funktion ist. Von einem Funktions-Knoten gehen zwei Kanten aus: eine verweist auf den Maschinencode des Rumpfes (siehe Abschnitt 3.5), die zweite zeigt auf ein Feld, das die Werte der freien Variablen des Rumpfes als Referenzen auf andere Knoten des Graphen aufnimmt. Wir werden auf die Darstellung von Funktionen im Maschinenmodell in den folgenden Abschnitten dieses Kapitels noch ausführlich zurückkommen, so daß wir an dieser Stelle nicht näher auf die Bedeutung dieser Komponenten eingehen wollen.

Abbildung 3.3 zeigt die Darstellung einer Funktion im Graphen. Wir haben hier eine graphische Repräsentation gewählt, die der Realisierung in der konkreten Implementierung bereits sehr nahe kommt. Wie bereits erwähnt, werden wir uns aber dennoch in diesem Kapitel noch nicht näher mit dem Aspekt des Speicherlayouts der verschiedenen Datenstrukturen beschäftigen. Wir werden nur bei Abbildungen auf eine uns als am geeignetesten erscheinende Darstellung zurück-

Abbildung 3.3 Darstellung von Funktions-Knoten



greifen, die auch beispielsweise Felder verwendet und nicht nur Knoten und Kanten benutzt.

3.4 Umgebungen und Register

Die Maschine führt die Instruktionen aus, indem sie diese wie im Graphenmodell relativ zu einer Umgebung interpretiert. Diese Umgebungen werden in der Maschine als *Register* bezeichnet. Register sind Felder, deren Elemente auf Knoten im Graph verweisen. Eine Schwäche des Graphenmodells aus Sicht der Implementierung liegt in der Tatsache, daß der Zugriff auf Umgebungen lediglich über die Namen von Variablen erfolgt, was in der Praxis mit einem Hashing-Schritt verbunden ist. Dieses Manko beseitigt der Compiler, indem er bereits zur Übersetzungszeit bei der *Registerallokation* jeder Variablen x des Quellprogramms auf einen Index i eines Registers abbildet, so daß die Maschine nun nur noch effizient über Indizes und nicht mehr über Namen auf die Werte von Variablen zugreifen kann.

Eine weitere Schwäche des Graphenmodells besteht darin, daß bei der Ausführung von Anweisungen die Umgebungen häufig erweitert, eingeschränkt oder kopiert werden müssen. Auch die Maschine kann dynamisch neue Registersätze erzeugen. Um dem Übersetzer allerdings eine dedizierte Einflußnahme auf die Allokation von Registern zur Laufzeit zu ermöglichen, unterstützt die Maschine unterschiedliche Arten von Registern mit unterschiedlich langer Lebensdauer. Dem Compiler fällt somit die Aufgabe zu, für jede Variable x des Quellprogramms eine Klassifikation vorzunehmen, indem er x eine Registerart (auch Adressierungsart genannt) zuweist und zudem auch noch einen Index i innerhalb des Registersatzes von x bestimmt.

Tabelle 3.1 faßt die verschiedenen Arten von Registern zusammen, die die Maschine unterstützt. Wir werden zunächst die Art der Verwendung dieser Register nur kurz skizzieren, werden in den folgenden Abschnitten dieses Kapitels aber genauer darauf eingehen.

Jeder Thread verfügt über einen Satz von n *allgemeinen* (engl. *general purpose*) Registern, mit festem n. Der Compiler kann diese auch *X-Register* genannten Register zum Zwischenspeichern temporärer Werte oder für sehr kurzlebige Variablen verwenden. Zudem werden sie für die Parameterübergabe bei der Funktionsapplikation, die Rückgabe von Werten bei Funktionsaufrufen und auch noch für verschiedene andere Zwecke verwandt, wie weiter unten beschrieben.

Tabelle 3.1 Registerarten der Maschine

Name	Lebensdauer	Schreibweise	Typische Verwendung
X	Thread	$\mathtt{X}_i$	temporäre Werte, Parameterübergabe, etc.
lokal	Funktionsinkarnation	$\mathtt{L}_i$	lokale Variablen einer Funktion
global	Funktion	$\mathtt{G}_i$	globale (= freie) Variablen einer Funktion
virtuell	Programm	$\mathtt{V}_i$	konstante Argumente

Zum Zugriff auf die lokalen Variablen einer Funktion dienen die *L-Register*. Der Compiler sorgt dafür, daß der Code, der für den Rumpf einer Funktion f erzeugt wird, bei Bedarf bei jedem Aufruf von f einen neuen Satz L-Register hinreichender Größe alloziert und diese entsprechend füllt.

Die *G-Register* dienen zur Adressierung der freien Variablen einer Funktion.

Jedes Programm verfügt über beliebig viele *virtuelle* oder *V-Register*, die zur Adressierung von konstanten Werten dienen, also Werten, die sich über die gesamte Lebensdauer eines Programmes hinweg nicht verändern werden. Daher kann bereits der Lader Referenzen auf diese Register auflösen, was den Namen „virtuell“ erklärt. Näheres hierzu siehe Abschnitt 3.5

Mit dem Begriff „Register“ bezeichnet man üblicherweise besonders ausgezeichnete, feste Speicherbereiche einer Maschine, auf die besonders effizient zugegriffen werden kann. In diesem Sinne verdient allerdings keines der oben beschriebenen Register der Maschine diesen Namen, da alle diese Register dynamisch mehrfach in wechselnden Speicherbereichen alloziert werden können. Wir schließen uns bei der Namensgebung dabei Warren [War77] an und verwenden den Begriff in der erweiterten Form. Neben diesen Registern, die ausschließlich zur Adressierung von Variablen verwandt werden, verwendet die virtuelle Maschine noch einige weitere Register, die nun aber im eigentlichen Sinne als „Register“ bezeichnet werden können: sie werden zum Beispiel für die Case-Anweisung oder zum Aufbau von Tupeln verwandt. Wir wollen aber an dieser Stelle nicht weiter vorgreifen, sondern werden diese Register im folgenden nach und nach einführen.

Die Qualität mit der der Compiler die Aufgabe der Registerallokation erfüllt, hat einen wichtigen Einfluß auf die Qualität des erzeugten Codes. Wir werden daher in einem eigenen Abschnitt (5.6) noch einmal gesondert auf diese Thematik im Zusammenhang mit L eingehen.

3.5 Maschinenprogramme

Wie bereits oben erwähnt, erzeugt der Übersetzer als Ausgabe ein Maschinenprogramm, das als Eingabe für den Lader dient. Bei einem Maschinenprogramm handelt es sich allerdings nicht lediglich um eine Folge von Maschineninstruktionen, sondern es besitzt mehr Struktur, wie Abbildung 3.4 zeigt: es besteht aus zwei Hauptteilen, der *Prelude* und einer Folge von (mindestens einem) *Segmenten*.

Abbildung 3.4 Format von Maschinenprogrammen

prog	::=	*prelude* seg^+	*Programm*
prelude	::=	`prelude` $pinstr^*$ `end`	*Prelude*
pinstr	::=	`vnewInt` $\mathtt{V}_i$ n	*Preludeinstruktionen*
	|	`vnewVar` $\mathtt{V}_i$	
	|	`vnewCon` $\mathtt{V}_i$	
	|	`import` $\mathtt{V}_i$ *string*	
	|	`export` $\mathtt{V}_i$ *string*	
	|	`vmakeHT` $\mathtt{V}_i$ n	
	|	`vaddHTCon` $\mathtt{V}_i$ $\mathtt{V}_k$ *label*	
	|	`vaddHTTuple` $\mathtt{V}_i$ $\mathtt{V}_k$ *label label*	
seg	::=	`segment` $\mathtt{S}_n$ $instr^*$ `end`	*Segmentdefinition*
instr	::=	[*label*:] *opcode* arg^*	*Maschineninstruktion*
opcode	::=	`unify` | `move` | `...`	
arg	::=	n	*Zahl*
	|	*label*	
	|	$\mathtt{X}_n$ | $\mathtt{L}_n$ | $\mathtt{G}_n$ | $\mathtt{V}_n$	*Register*
	|	$\mathtt{S}_n$	*Segmentreferenz*
label	::=	*string*	*Sprungziel*

3.5.1 Prelude

Die Instruktionen der virtuellen Maschine sollen die Möglichkeit haben, direkt auf Knoten im Speicher zu verweisen. Damit dies möglich wird, müssen diese Knoten zuerst im Speicher aufgebaut werden, bevor die Instruktionen selbst geladen werden. Diese Aufgabe übernimmt die Prelude.

Die Prelude besteht aus einer Folge von wenigen speziellen *Preludeinstruktionen*, die nur hier verwandt werden können. Diese Instruktionen werden anders als die Instruktionen der Segmente bereits zur Ladezeit direkt vom Lader selbst ausgeführt. Ihre Aufgabe besteht in der Initialisierung der V-Register. Die Funktionsweise der einzelnen Instruktionen wird wie folgt beschrieben:

`vnewInt` $\mathtt{V}_i$ n
Die Instruktion erzeugt einen neuen Zahl-Knoten mit Wert n im Speicher und legt eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$ ab.

`vnewVar` $\mathtt{V}_i$
Erzeugt einen neuen Variable-Knoten im Speicher und speichert eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$.

`vnewCon` $\mathtt{V}_i$
Erzeugt einen neuen Konstruktor-Knoten im Speicher und speichert eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$.

Die Instruktionen `vmakeHT`, `vaddHTCon` und `vaddHTTuple` werden zur Erzeugung von Hashtabellen für die Indizierung von Case-Ausdrücken benötigt und werden in Abschnitt 4.6.4 beschrieben.

Die letzten beiden Instruktionen `import` und `export` werden zur getrennten Übersetzung benötigt und werden in Abschnitt 3.10 beschrieben.

3.5.2 Segmente

Auf die Prelude folgen ein oder mehrere Segmente, die die eigentlichen Maschineninstruktionen eines Programms enthalten. Der Compiler legt den Code, den er für den Rumpf jeder Funktionsdefinition erzeugt, jeweils in einem eigenen Segment ab. Schließlich wird der ganze verbleibende Toplevel (also alles bis auf die in einem Programm enthaltenen Funktionsrümpfe) in einem eigenen und zwar immer in dem ersten Segment abgelegt; das ist wichtig, da wir in Abschnitt 3.5.4 sehen werden, daß der Lader sicherstellt, daß nach dem Laden immer das erste Segment eines Programmes zur Ausführung gebracht wird.

Jedes Segment besteht aus einer Folge von Maschineninstruktionen, die jeweils aus dem Namen der Instruktion gefolgt von (eventuell 0) Argumenten bestehen. Dabei kann jede Instruktion optional noch mit einer Zeichenkette als Marke für ein Sprungziel versehen werden. Ein Argument einer Instruktion kann entweder eine Zahl n, ein symbolisches Sprungziel oder eine Referenz auf ein Register in der Form $\texttt{X}_i$, $\texttt{L}_i$, $\texttt{G}_i$ oder $\texttt{V}_i$ sein. Nur die Instruktion zur Funktionserzeugung erlaubt als einzige Instruktion auch Referenzen auf andere Segmente in der Form $\texttt{S}_i$; dabei ist allerdings sichergestellt, daß $\texttt{S}_i$ stets auf ein Segment des gleichen Programms verweist.

3.5.3 Beispiel

Abbildung 3.5 zeigt links ein kleines Beispielprogramm (das lediglich zur Illustration dient und daher wenig sinnvolle Anweisungen enthält) und rechts ein mögliches virtuelles Maschinenprogramm dazu (der Überschaubarkeit wegen wurde hier auf Optimalität des Maschinenprogramms kein Wert gelegt). Wir wollen an dieser Stelle noch nicht genau auf jede der hier verwandten Instruktionen eingehen, da diese erst in den folgenden Abschnitten dieses Kapitels beschrieben werden, sondern wollen uns hier darauf konzentrieren, die Zusammenhänge zwischen Prelude und Segmenten zu verdeutlichen.

Das gesamte Programm besteht aus insgesamt 3 Segmenten: $\texttt{S}_0$ für den Toplevel und $\texttt{S}_1$ und $\texttt{S}_2$ für die Rümpfe der Funktionen `f` und `g` respektive.

Die Prelude initialisiert zunächst 2 V-Register: der Konstruktor `ABC` kommt nach $\texttt{V}_0$, während $\texttt{V}_2$ die Zahl 5 aufnimmt. Somit können Referenzen auf die Variable `ABC` direkt über ein V-Register effizient aufgelöst werden, wie dies bei der Unifikation in Segment $\texttt{S}_1$ geschehen ist.

Das Toplevel Segment $\texttt{S}_0$ erzeugt zunächst eine Funktion unter Verwendung der Instruktion `fun`, die wir in Abschnitt 3.8.3 genauer beschreiben werden, daher hier nur soviel: das erste Argument $\texttt{X}_1$ bezeichnet das Register in das die neu erzeugte Funktion geschrieben wird; das letzte Argument $\texttt{S}_1$ gibt an, in welchem Segment, der Code, der für den Rumpf der Funktion `f` erzeugt wurde, zu finden ist. Die letzte Anweisung `return` in $\texttt{S}_0$ markiert das Ende einer Anweisungssequenz und wird in Abschnitt 3.5.5 näher erläutert.

Der Rumpf von `f` in Segment $\texttt{S}_1$ erzeugt ebenfalls zunächst eine neue Funktion mittels der

Abbildung 3.5 Beispielprogramm und zugehöriger virtueller Maschinencode

```
let
  con ABC;
  fun f(x) =
    let fun g() = 5
    in
      spawn(g);
      unify(x,ABC);
      x
    end
in
  f
end
```

```
prelude
  vnewCon V0          % ABC
  vnewInt V1 5
end

segment S0            % Toplevel
  fun X1 0 0 S1
  return X1
end

segment S1            % f
  fun X1 0 0 S2
  spawn X1
  unify X0 V0
  return X0
end

segment S2            % g
  return V1
end
```

`fun` Anweisung. Anschließend wird dann mittels der `spawn` Instruktion (näheres siehe Abschnitt 3.8.1) ein neuer Thread erzeugt; das Argument von `spawn` bezeichnet dabei das Register, indem die nullstellige Funktion zu finden ist, die in dem neuen Thread appliziert werden soll.

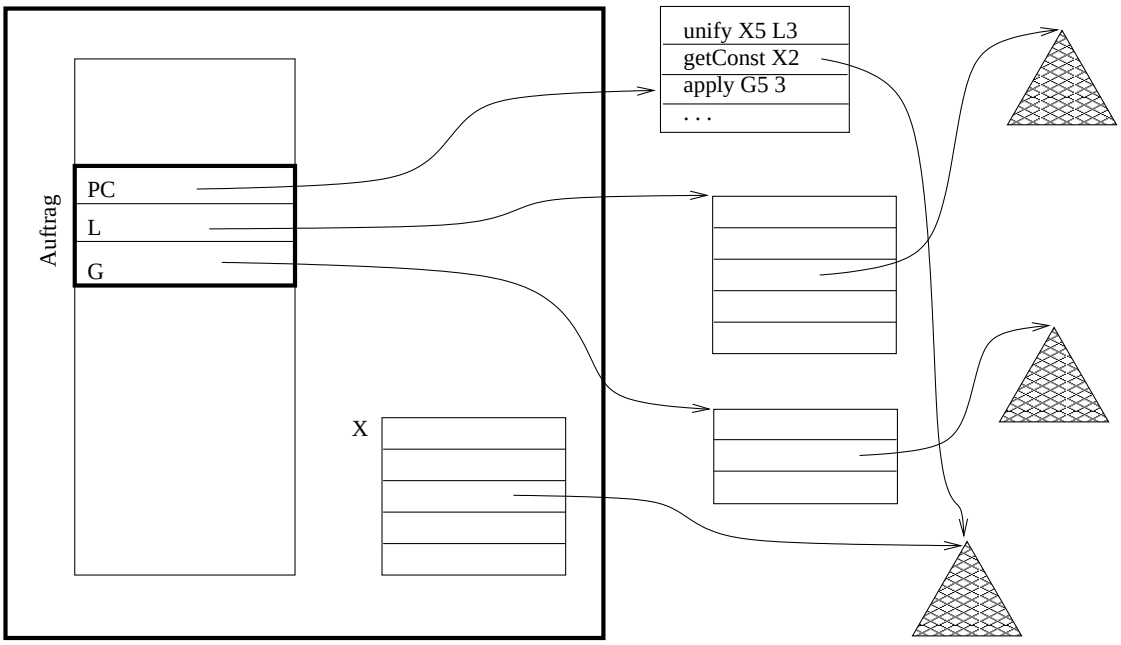
3.5.4 Der Lader

Wie bereits weiter oben erwähnt besteht die Aufgabe des Laders darin, ein virtuelles Maschinenprogramm in eine adäquate interne Repräsentation zu transformieren. Dabei wird in 3 Schritten vorgegangen.

1. Aufbau der V-Register
2. Laden der Segmente:
   (a) Auflösen von Referenzen auf V-Register durch deren Inhalte
   (b) Auflösen symbolischer Segmentreferenzen
   (c) Auflösen symbolischer Sprünge
3. Ausführung des ersten Segments

Zunächst werden die Inhalte der V-Register bestimmt: das geschieht durch Ausführung der Instruktionen der Prelude. Daran anschließend werden die Segmente geladen; dabei werden symbolische Referenzen auf andere Segmente und auf Sprungziele aufgelöst. Am wichtigsten ist aber die Auflösung von Referenzen auf V-Register: die Instruktionen der Segmente können nur lesend und nicht schreibend auf V-Register zugreifen. Da alle Inhalte der V-Register beim Laden der Segmente bereits bekannt sind (da die Prelude bereits vollständig ausgeführt wurde),

Abbildung 3.6 Zusammenhang Thread, Auftrag, Register, Instruktionen



kann der Lader nun die indirekten Referenzen auf V-Register in den Instruktionen durch direkte Referenzen auf die entsprechenden Knoten des Speichers ersetzen. Das erlaubt dann beim späteren Ausführen des Codes einen schnelleren Zugriff auf diese Argumente, da eine Indirektion eingespart wurde. So gesehen sollte der Compiler versuchen, möglichst viele V-Register zu verwenden; für Funktionen, die nicht innerhalb von anderen Funktionen deklariert wurden, könnte man prinzipiell sogar ganz auf die Verwendung von G-Registern verzichten und nur V-Register benutzen. In der Praxis ist dies allerdings nicht möglich: so gibt es viele Instruktionen, die in ihren Argumenten zwar die Verwendung von X-, L- und G-Registern erlauben nicht aber die von V-Registern. Der Grund hierfür liegt in der Darstellung der Instruktionen in der Implementierung, auf die wir in Abschnitt 6.5 näher eingehen: um den Dekodieraufwand zu reduzieren, kodiert man die Registerarten der Argumente in den Operationscode der Instruktion und erhält somit aus einer Instruktion wieder mehrere verschiedene Instanzen. Um nun den Instruktionssatz klein zu halten, verbietet man für viele Instruktionen die Verwendung von V- und zum Teil auch von G- oder sogar L-Registern.

Abschließend stellt der Lader sicher, daß das Programm auch zur Ausführung kommt: das geschieht, indem der Lader auf einem dafür erzeugten Thread einen Abschluß (mit leerer Umgebung, siehe Abschnitt 3.5.5) ablegt, so daß das erste Segment des gerade geladenen Programms zur Ausführung kommt.

3.5.5 Threads und Aufträge

Im Graphenmodell bearbeitet jeder Thread genau einen Abschluß, der Ausdrücke enthält, die wiederum selbst wieder Unterausdrücke enthalten. Im Maschinenmodell wird nun jeder Aus-

druck in eine Sequenz von Instruktionen übersetzt. Bei einem Funktionsaufruf etwa in der Form

$$\texttt{let val } x = f(y) \texttt{ in } e \texttt{ end}$$

entsteht nun die Situation, daß ein neuer Unterausdruck (der Rumpf der Funktion f) entsteht, so daß der noch auszuwertende Ausdruck e gemerkt werden muß. In der Maschine verwenden wir dazu die etablierte Technik, die einen Thread als *Stapel* von Abschlüssen darstellt. Der oberste Abschluß ist derjenige, den der Thread gerade auswertet. Bei einer Funktionsapplikation wird dann zuoberst auf dem Thread der Rumpf der Funktion, die appliziert wird, zusätzlich abgelegt.

Im Maschinenmodell bestehen allerdings die Umgebungen der Abschlüsse im Gegensatz zum Graphenmodell aus zwei Komponenten. Auch enthalten sie nicht die Werte für alle Variablen der zugehörigen Anweisung: wie zuvor bereits erwähnt, können manche Variablen über X-Register angesprochen werden. Wir sprechen daher nicht mehr von Abschlüssen, sondern bezeichnen die Einträge auf dem Stapel eines Threads als *Aufträge* (engl. *task*) (vgl. Abbildung 3.6).

Ein Auftrag ist ein Tripel der Form $\langle$`PC`, `L`, `G`$\rangle$. Dabei ist `PC` eine Referenz auf eine Sequenz von Maschineninstruktionen, der sogenannte *Programmzähler*. Die Zeiger `L` und `G` bilden die Umgebung; auf ihre Bedeutung werden wir in Abschnitt 3.8 näher eingehen. Die Maschine führt stets den obersten Auftrag eines Threads aus. Das Löschen eines Auftrags geschieht explizit durch Ausführung einer speziellen Instruktion (siehe unten).

Die Ausführung eines Auftrages bedeutet die Ausführung der ersten Instruktion I, auf die `PC` verweist. Dabei werden Referenzen von I auf X-, L- und G-Register über die X-Register des aktuellen Threads und die `L` und `G` Zeiger des Auftrages aufgelöst. Die Ausführung einer Instruktion kann die Zeiger `PC` und `L` des aktuellen Auftrages modifizieren, es gibt keine Instruktion, die `G` ändert. Einige Instruktionen bewirken das Löschen des obersten Auftrages oder die Erzeugung eines neuen Auftrages. Dies ist die einzige Möglichkeit auf einen anderen G-Registersatz umzuschalten.

Nach der Ausführung von I zeigt `PC` auf die auf I folgende Instruktion, beziehungsweise (falls es sich bei I um einen bedingten oder unbedingten Sprungbefehl handelt) auf das Sprungziel.

Zeigt `PC` auf eine leere Sequenz, so wird dieser Auftrag vom Thread genommen und der darunter liegende Auftrag ausgeführt. In der Praxis ist es allerdings zu teuer, nach der Ausführung jeder Anweisung zu überprüfen, ob noch eine weitere in der Sequenz existiert. Die Maschine stellt stattdessen eine eigene Instruktion `return` hierzu zur Verfügung. Diese markiert das Ende einer Sequenz und ist die einzige Instruktion, die bewirkt, daß der aktuelle Auftrag vom Stapel des Threads genommen wird. Der Compiler muß nun sicherstellen, daß `PC` nie auf eine leere Sequenz zeigen kann (was einfach erreicht werden kann, indem beispielsweise am Ende jedes Segments eine `return` Anweisung eingesetzt wird).

3.6 Signalbehandlung

Bei der Bearbeitung von Aufträgen muß die Maschine in regelmäßigen Abständen das Eintreten verschiedener Ereignisse prüfen und abhängig davon eine *Signalbehandlung* durchführen.

Mögliche Signale sind beispielsweise:

Preemption Die Zeitscheibe des aktuellen Threads ist abgelaufen oder ein Thread mit höherer Priorität wurde geweckt oder erzeugt.

Speicherbereinigung Ein bestimmter Schwellwert beim Speicherverbrauch wurde überschritten, so daß eine Speicherbereinigung nötig wird.

Ein/Ausgabe Auf einem Eingabepuffer liegen neue Daten an oder ein zuvor vollständig gefüllter Ausgabepuffer kann jetzt wieder neue Daten aufnehmen.

Bei der Behandlung von Signalen muß unbedingt die Fairneßeigenschaft beachtet werden: eine nichtterminierende Berechnung darf die Signalbehandlung nicht verhindern. Eine Möglichkeit wäre daher, diese vor der Ausführung jeder einzelnen Instruktion durchzuführen;[2] das ist allerdings aus praktischer Sicht zu teuer. Außerdem ist das Design der Maschine so angelegt, daß bestimmte Folgen von Instruktionen nicht unterbrochen werden können; das gilt zum Beispiel für die Anweisungen des Wächters eines Case. Für schwache Realzeitanforderungen (engl. soft realtime) reicht es aus, nur bei solchen Instruktionen eine Signalbehandlung durchzuführen, die auch in nichtterminierenden Berechnungen sicher nach endlicher Zeit ausgeführt werden. Eine solche Instruktion ist `apply`, die für die Funktionsapplikation erzeugt wird (siehe Abschnitt 3.8.3).

Zur Realisierung von Case werden wir in Abschnitt 3.9 auch unbedingte Sprünge einführen. Damit ist es möglich, eine Endlosschleife zu realisieren, die keinen Funktionaufruf enthält (zum Beispiel durch eine Instruktion, die immer auf sich selbst springt). Der Compiler verwendet Sprünge allerdings nur eingeschränkt: ein Sprung kann nur auf eine Instruktion erfolgen, die im gleichen Segment hinter der aktuellen Instruktion liegt. Es wird also nur vor und nie zurück gesprungen. Da alle Segmente endliche Länge haben, ist somit die Fairneßeigenschaft gesichert.

Die Maschine testet somit genau vor jeder Funktionsapplikation, ob eine Signalbehandlung notwendig ist. Wenn ja, wird der gerade bearbeitete Thread T angehalten, das heißt er wechselt vom Zustand laufend in den Zustand ausführbar. Dazu sind keine besonderen Maßnahmen (wie etwa das Retten bestimmter Register der Maschine) nötig, da T an dieser Stelle alle zu seiner Ausführung benötigten Daten bereits in sich enthält. T wird somit einfach in die Liste der angehaltenen Threads aufgenommen und die Signalbehandlung wird durchgeführt.

Es bleibt noch zu klären, wie die Zeitscheibe realisiert wird. Hierzu besteht zum einen die Möglichkeit, einen Zähler zu verwalten, der während der Laufzeit eines Threads nach und nach inkrementiert wird. Sobald ein gewisser Schwellwert überschritten ist, kann dann eine Threadumschaltung durchgeführt werden. Dieses Verfahren verursacht allerdings zusätzliche Laufzeitkosten für die Verwaltung des Zählers. Wesentlich praktikabler ist es, von den Möglichkeiten moderner Betriebssysteme und Prozessoren Gebrauch zu machen: diese bieten die Möglichkeit, innerhalb eines Benutzerprozesses in festen Zeitintervallen (mit einer Auflösung im Millisekundenbereich) eine frei wählbare Prozedur p aufzurufen. Die Maschine implementiert p dann so, daß p lediglich ein bestimmtes Flag in der Maschine setzt, das dann später bei Ausführung von `apply` getestet wird.

3.7 Suspension

Muß die Abarbeitung einer Instruktion I suspendieren, kann also I wegen fehlender Information im Speicher nicht bearbeitet werden, so muß der zu I gehörende Thread T suspendiert werden. Bei Suspension gilt, daß I grundsätzlich nicht verändert wird; T hält also auf der Instruktion I an und schaltet nicht wie sonst üblich auf die auf I folgende Instruktion. Dadurch wird gewähr-

[2] Wir gehen davon aus, daß die Ausführung jeder einzelnen Instruktion stets nur endlich viel Zeit benötigt.

leistet, daß T nach dem Wecken diejenige Instruktion, die die Suspension veranlaßt hat, noch einmal ausführt.

T muß dann wieder geweckt werden, sobald I ausführbar wird. Das ist prinzipiell dann möglich sobald neue Information zum Speicher hinzukommt. Es ist allerdings nicht praktikabel I nach *jeder* Änderung des Speichers auf Ausführbarkeit zu testen.

Das Constraintsystem von L über rationalen Bäumen verwendet (wie auch alle anderen in Mozart integrierten Constraintsysteme) eine *variablenzentrierte* Darstellung: das bedeutet, daß die Information im Speicher direkt mit einzelnen Variablen assoziiert ist. Demzufolge kann die Suspension einer Instruktion I nur dann auftreten, wenn die im Speicher enthaltene Information über eine bestimmte endliche Menge von Variablen $\overline{x}$ noch nicht stark genug ist. Somit muß eine Instruktion dann wieder auf Ausführbarkeit getestet werden, sobald im Speicher neue Constraints zu *mindestens einer* Variable x aus $\overline{x}$ hinzugefügt werden, was für L wiederum gerade bedeutet, daß x gebunden wurde.

3.7.1 Suspensionslisten und Suspensionen

Diese Überlegungen geben bereits eine erste Idee für eine mögliche Realisierung: jeder Variable-Knoten im Speicher hat eine Referenz auf eine *Suspensionsliste*, das ist die Liste aller Threads, die auf diese Variable suspendieren. Umgekehrt ist es nicht notwendig, daß ein Thread Referenz auf alle Variablen erhält, auf die er suspendiert. Beim Binden einer Variablen x werden dann einfach alle Threads aus der Suspensionsliste von x geweckt.

Eine Schwierigkeit gilt es allerdings noch zu meistern, die dadurch zustande kommt, daß ein Thread auf mehr als nur eine Variable suspendieren kann: suspendiert ein Thread T beispielsweise auf zwei Variablen x und y, so wird T sowohl in die Suspensionsliste von x als auch in die von y eingetragen. Wird nun zunächst x gebunden, dann wird T geweckt, aber nicht unbedingt auch direkt bearbeitet, weil unter Umständen noch andere Threads zuvor bedient werden müssen. Wird nun y gebunden, bevor T zur Ausführung gebracht wird, so muß einfach zu erkennen sein, daß T bereits geweckt wurde.

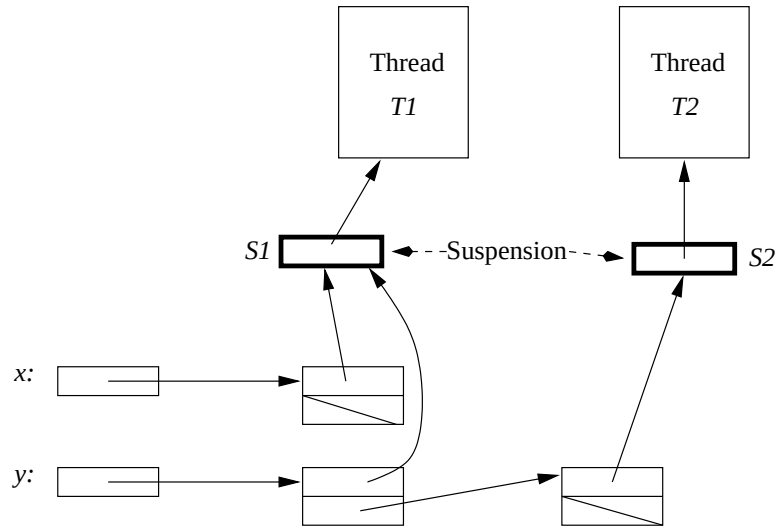
Zur Lösung dieses Problems führen wir eine Indirektion ein: die Suspensionslisten der Variablen verweisen nicht direkt auf Threads sondern auf einen speziellen *Suspensionsknoten*, der dann erst auf den richtigen Thread verweist.

Wir wollen das Vorgehen am Beispiel erklären: ausgehend von zwei ungebundenen Variablen x und y zeigt Abbildung 3.7 den Speicherzustand, bei der Suspension zweier Threads T_1 und T_2. Dabei suspendierte in T_1 die Ausführung einer Addition $x + y$ auf *beide* Variablen, während T_2 bei Ausführung von `wait(`y`)` lediglich auf y suspendiert. Dem entsprechend taucht T_1 in Abbildung 3.7 sowohl in der Suspensionliste von x als auch in der von y auf, allerdings nur indirekt, indem nämlich die entsprechenden Listenelemente von x und y auf den *gleichen* Suspensionsknoten S_1 verweisen, der dann erst auf T_1 zeigt.

Wir wollen nun betrachten, was im Einzelnen passiert, wenn ein dritter Thread die Anweisung `unify(`x`,5)` ausführt: dies bewirkt ein Binden von x, wodurch der einzige Thread T_1 in der Suspensionsliste von x geweckt wird. Wichtig ist nun aber, daß zusätzlich die Referenz von S_1 auf T_1 gelöscht wird, wodurch ein weiteres Wecken über y verhindert wird.

Wenn T_1 dann von der Maschine bearbeitet wird, wird zunächst wieder versucht, die Addition $x + y$ auszuführen. Dabei wird dann erkannt, daß x gebunden ist, daß aber y immer noch auf

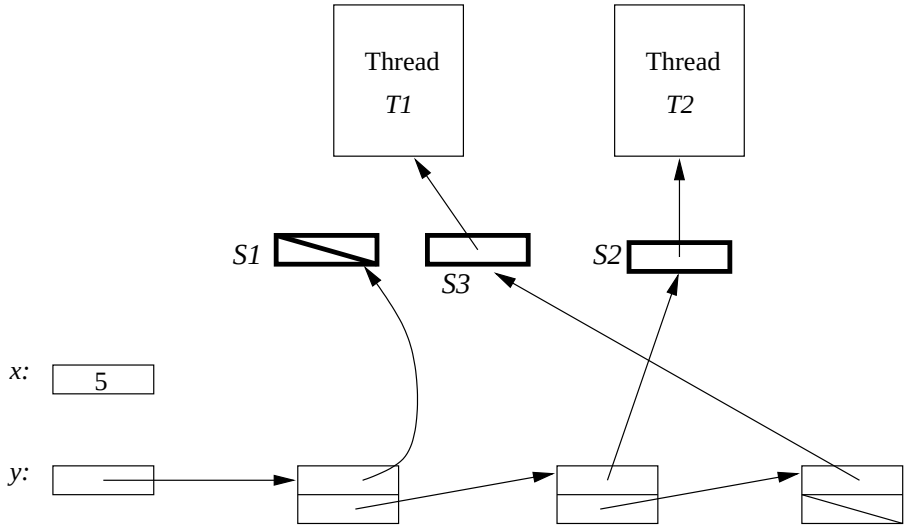
Abbildung 3.7 Thread T_1 suspendiert auf $x + y$, T_2 suspendiert auf `wait(`y`)`.



einen Variable-Knoten verweist. Jetzt wird ein *neuer* Suspensionsknoten S_3 erzeugt, der auf T_1 verweist und dieser in die Suspensionsliste von y zu den bereits existierenden Verweisen auf S_1 und S_2 eingefügt. Abbildung 3.8 zeigt den Inhalt des Speichers zu diesem Zeitpunkt.

Wenn nun schließlich noch `unify(`y`,3)` ausgeführt wird, passiert folgendes: die Suspensionsliste von y wird durchlaufen und alle Suspensionen, die noch nicht geweckt wurden (die also noch auf einen Thread verweisen), werden geweckt. Im Beispiel sind das S_2 und S_3, so daß T_1 und T_2 geweckt werden, wodurch jetzt $x+y$ beziehungsweise `wait(`y`)` nun ohne wieder zu suspendieren bearbeitet werden können.

Abbildung 3.8 Situation nach Ausführen von `unify(`x`,5)`, Wecken von T_1 und neuerlicher Suspension von T_1



3.7.2 Suspension auf Gleichheit zweier Variablen

Bei der Suspension bedarf ein Fall besonderer Beachtung, nämlich der, daß ein Thread auf die Gleichheit zweier Variablen suspendiert, was zum Beispiel dann der Fall ist, wenn in folgendem Ausdruck

```
case [x,y] of
   [z,z] => ...
```

weder x noch y gebunden sind. Die Maschine wird dann bei der Ausführung der Unifikation im Wächter eine der beiden Variablen an die andere binden. Nehmen wir beispielsweise an, daß x an y gebunden wird. Im Abschnitt über Case (3.9) werden wir sehen, daß nur eine Suspension in die Suspensionsliste der gebundenen Variable, also x, eingetragen wird. Wenn nun später ein anderer Thread eine Unifikation von x mit y durchführt (außerhalb eines Wächters, so daß dies global sichtbar wird), kommt es nun darauf an, in welcher Richtung die Maschine die Bindung vornimmt. Wenn jetzt in umgekehrter Richtung gebunden wird, also y an x, dann werden wie oben beschrieben nur die Suspensionen von y geweckt, so daß das suspendierte Case fälschlicherweise nicht wieder geweckt wird.

Grundsätzlich gibt es verschiedene Möglichkeiten, diese Problem in den Griff zu bekommen:

- Bei Suspension auf Gleichheit zweier Variablen x und y wird die gleiche Suspension sowohl in x als auch in y eingetragen. Beim späteren Binden ist die Bindungsrichtung dann egal. Wie oben beschrieben, kommt die Maschine auch mit der Tatsache zurecht, daß die gleiche Suspension in der Suspensionliste zweier verschiedener Variablen vorkommt.
- Die Maschine stellt sicher, das bei einer Unifikation zweier Variablen immer in die gleiche Richtung gebunden wird. Das setzt voraus, daß auf Variable-Knoten eine totale Ordnung definiert werden kann, die effizient zu testen ist. Die Standardtechnik hierzu ist als Ordnungskriterium, die Lage im Speicher (das heißt die Adresse des Knotens) heranzuziehen. Das verkompliziert aber die Speicherbereinigung deutlich: diese muß dann nämlich sicherstellen, daß sich die relative Lage von Variablen zueinander nicht ändert.
- Bei einer Unifikation zweier Variablen werden *beide* Suspensionslisten geweckt. Das hat aber den Nachteil, daß die meisten Threads, die sich in der Suspensionsliste derjenigen Variablen befinden, die nicht gebunden wurde, in der Regel unnötig geweckt werden und dann direkt wieder von Neuem suspendieren müssen.

Aufgrund der genannten Nachteile der beiden letzten Varianten entschieden wir uns für die erste Alternative.

3.8 Instruktionen

Wir wollen im folgenden die verschiedenen Instruktionen der Maschine beschreiben. Die Implementierungsmuster und Instruktionen, die wir im folgenden vorstellen, decken stets den allgemeinsten Fall ab und sind noch nicht immer optimal. Wir wollen uns aber in diesem Abschnitt zunächst darauf beschränken, die grundsätzlichen Ideen zu verdeutlichen. Wir werden dann in Kapitel 4 darauf zurückkommen und verschiedene Techniken zur Optimierung vorstellen.

Jeder Ausdruck e wird in eine Folge von Instruktionen übersetzt. Da die Auswertung eines Ausdruckes e auch immer einen Wert liefert, sorgt der Maschinencode dafür, daß der Wert von e in einem Register abgelegt wird. Dabei handelt es sich nicht um ein spezielles Register, sondern der Code kann selbst bestimmen, in welches Register der Wert eines Ausdruckes geschrieben wird. So kann beispielsweise für einen Ausdruck der Form

$$\texttt{let val } x = e \texttt{ in } \ldots \texttt{ end}$$

der Maschinencode für e gleich so ausgelegt werden, daß er den Wert von e in dem Register ablegt, das der Compiler für x vorgesehen hat.

Die Sprache L erlaubt in Ausdrücken an fast allen Stellen beliebige Subausdrücke. So kann eine Funktionsapplikation im allgemeinen wie folgt aussehen:

$$e_0(e_1, \ldots, e_n)$$

Bei der Übersetzung eines solchen Ausdruckes nimmt der Compiler eine Vereinfachung vor, indem er komplexe Subausdrücke durch die Einführung von Hilfsvariablen simplifiziert. So wird die Applikation von oben intern im Compiler nach

```
let val x_0 = e_0;
    val x_1 = e_1;
    val ···
    val x_n = e_n
in
  x_0(x_1, ..., x_n)
end
```

transformiert, was dann eine einfachere Behandlung innerhalb des Compilers erlaubt. Bei der Beschreibung von Übersetzungsmustern für die verschiedenen Ausdrücke von L gehen wir daher im folgenden davon aus, daß Ausdrücke an bestimmten Stellen nur Variablen enthalten, wo im allgemeinen beliebige Ausdrücke erlaubt sind.

Zur besseren Strukturierung nehmen wir eine Gruppierung anhand der Quellsprache vor und beschreiben, welche Maschineninstruktionen zur Umsetzung der einzelnen L Ausdrücke benötigt werden. In jedem Abschnitt werden wir dabei zunächst die verschiedenen neuen Instruktionen beschreiben und werden daran gegebenenfalls ein Beispiel zur Verdeutlichung anschließen.

Zur Notation schreiben wir im folgenden R für ein beliebiges X-, L-, G- oder V-Register und bezeichnen mit R_x das Register, das der Compiler der Variablen x zugeordnet hat. Konstanten in Form von Zahlen schreiben wir direkt, verwenden also beispielsweise

`unify` $\texttt{X}_9$ `4711` statt `unify` $\texttt{X}_9$ $\texttt{V}_{23}$

wenn die Zahl `4711` im virtuellen Register Nummer 23 abgelegt wurde.

Manche der im folgenden vorgestellten Instruktionen benötigen Zugriff auf die (im Speicher eindeutig dargestellten) Konstruktor-Knoten für **`true`**, **`false`**, **`unit`**, `::` und `nil`. Hierzu stellt die Maschine spezielle Register `truep`, `falsep`, `unitp`, `consp` und `nilp` zur Verfügung, die auf die entsprechenden Knoten verweisen.

3.8.1 Operatoren

Für jeden Operator wird eine eigene Maschineninstruktion eingeführt.

Zellen

Für Zellen werden die folgenden Instruktionen benötigt:

`newRef` R R'
: Erzeugt eine neue Zelle im Speicher und legt eine Referenz darauf in Register R ab. Inhalt der Zelle ist der Knoten, auf den R' verweist.

`exchange` R R' R''
: Die Instruktion wartet (das heißt der ausführende Thread wird solange suspendiert), bis R eine Zelle referiert. Danach wird in R' der aktuelle Inhalt der Zelle abgelegt und der neue Inhalt der Zelle zu R'' gesetzt.

Beispiel Die oben beschriebenen Instruktionen haben ihre direkte Entsprechung in Quellsprachenausdrücken; wir wollen daher hier nur ein kurzes Beispiel bringen. So wird

```
let val x = ref(23);
    val y = exchange(x,27)
in e
end
```

in eine Sequenz von Maschineninstruktion übersetzt:

`newRef` $R_{\texttt{x}}$ `23`
`exchange` $R_{\texttt{x}}$ `27` $R_{\texttt{y}}$
⟨*Code für* e⟩

Wir sind hier davon ausgegangen, daß sich die Zahlen 23 und 27 in virtuellen Registern befinden. Der Compiler muß also noch zusätzliche Instruktion `vnewInt` in der Prelude ganz zu Anfang des Programms generieren. Diese Instruktionen haben wir hier der Übersichtlichkeit halber weggelassen und werden auch im folgenden diese nicht mehr explizit erwähnen.

Konstruktoren und logische Variablen

Zur Erzeugung von Konstruktoren und logischen Variablen dienen die Instruktionen `newCon` und `newVar`:

`newCon` R
: Erzeugt einen neuen CON-Knoten im Speicher und legt eine Referenz darauf in Register R ab.

`newVar` R
: Erzeugt einen neuen VAR-Knoten im Speicher und legt eine Referenz darauf in Register R ab.

Arithmetik

Die Instruktionen für die arithmetischen Operatoren arbeiten analog zu den im vorangegangenen Abschnitt eingeführten Instruktionen.

`plus` R R' R''
: Die Instruktion wartet, bis R und R' auf einen Zahl-Knoten verweisen, erzeugt dann einen neuen Zahl-Knoten k im Speicher, der die Summe beider Werte enthält und legt anschließend eine Referenz auf k in R'' ab.

Die Instruktionen `minus`, `times`, `div` und `less` für die restlichen arithmetischen Funktionen sind analog definiert.

Bei `less` ist zu beachten, daß diese Instruktion einen der Konstruktoren **`true`** und **`false`** im Ausgaberegister zurückliefert und hierzu, wie zuvor beschrieben, die Register `truep` und `falsep` verwendet.

Unifikation

Zur Implementierung des Unifikations-Operators dient genau eine Instruktion:

`unify` R R'
: Die Instruktion unifiziert die Knoten, auf die die Register R und R' verweisen.

Thread-Erzeugung

Zur Erzeugung eines neuen Threads dient die folgende Anweisung:

`spawn` R
: Die Instruktion wartet, bis das Register R eine nullstellige Funktion f enthält. Sie erzeugt dann einen neuen Thread T, der genau einen Auftrag $\langle$**`nil`**, G, $PC\rangle$ enthält. Bei diesem bleibt der L-Registersatz leer, G ist eine Referenz auf den Abschluß von f und PC verweist auf den Anfang des Codesegmentes der Funktion f.

3.8.2 Deklaration

Eine Deklaration der Form

$$\textbf{let val } x = e_1 \textbf{ in } e_2 \textbf{ end}$$

wird so übersetzt, daß einfach die Codesequenz, die für e_2 erzeugt wird, an die Sequenz für e_1 angehängt wird. Dabei wird der Code für e_1 so angelegt, daß das Ergebnis von e_1 ins Register R_x geschrieben wird.

3.8.3 Funktionen

Im Zusammenhang mit Funktionen unterscheiden wir Instruktionen, die zur Erzeugung einer Funktion, zur Applikation und zur Übersetzung des Rumpfes dienen.

Funktionserzeugung

`fun` R n k S
: Erzeugt einen neuen Funktions-Knoten mit Arität n und k freien Variablen und legt eine Referenz darauf in R. Die Instruktion erwartet die Werte der globalen Variablen in den Registern $\texttt{X}_0$ bis $\texttt{X}_{k-1}$. S ist eine Referenz auf das Segment, das den Code für den Rumpf der Funktion enthält.

`move` R R'
: Die Instruktion kopiert den Inhalt von R in das Register R'.

Beispiel Wir wollen betrachten, wie der Code zur Erzeugung der rekursiven Funktion `f` aus folgendem Programmfragment aussieht (auf die Codeerzeugung für die Rümpfe von Funktionen gehen wir weiter unten ein):

```
...
let val f   = lvar();
    val aux = fn(x,y) => f(x+u,y+v) end
in
    unify(f,aux);
    ...
end
```

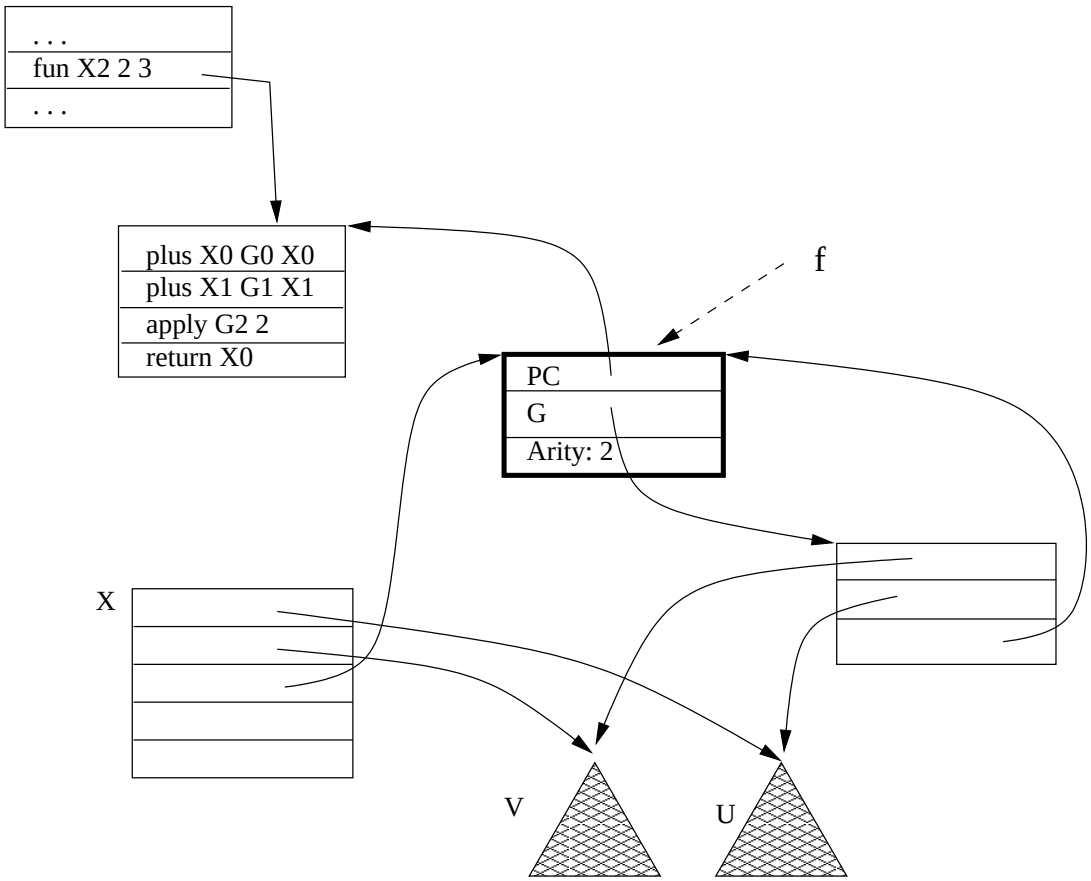
Hier wird eine Funktion `f` definiert. Da `f` rekursiv ist, wird deren Definition wie in Abschnitt 2.10.3 erläutert über die Einführung einer Hilfsvariable `aux` ermöglicht. Der Rumpf von `f` hat drei freie Variablen: neben `u` und `v` ist das auch `f` selbst. Die `fun` Instruktion erwartet nun die Werte der freien Variablen des Rumpfes in X-Registern. Der Maschinencode zu obigem Ausdruck sieht dann wie folgt aus:

```
...
move R_u X0
move R_v X1
newVar R_f
move R_f X2
fun X3 2 3 S_f
unify R_f X3
...
```

Hier werden zunächst Referenzen auf `u` und `v` nach $\texttt{X}_0$ und $\texttt{X}_1$ kopiert. Dann wird für `f` ein neuer Variable-Knoten erzeugt. Eine Referenz auf diesen Knoten kommt mittels `move` nach $\texttt{X}_2$. Das nächste freie X-Register ist $\texttt{X}_3$, das wir für die Variable `aux` verwenden. Die `fun` Instruktion schreibt dann die neu erzeugte Funktion nach $\texttt{X}_3$. Anschließend wird dann durch eine Unifikation von $\texttt{X}_3$ mit `f` ein Bindung von `f` an die neue Funktion erreicht.

Der Umweg bei rekursiven Funktionen über eine neue Variable und eine anschließende Unifikation zu gehen, läßt sich einsparen, wenn man die Reihenfolge der Arbeitsschritte der `fun`-Instruktion sorgfältig wählt: wenn `fun` zuerst einen neuen Funktions-Knoten erzeugt und diesen im Ausgaberegister ablegt, *bevor* die G-Registerbank aufgebaut wird, dann läßt sich der Code von oben deutlich einfacher und effizienter gestalten:

Abbildung 3.9 Speicherinhalt nach Erzeugung einer Funktion




```
move R_u X_0
move R_v X_1
fun X_2 2 3 S_f
move X_2 R_f
```

Hier wird also der neu erzeugte Funktions-Knoten zuerst in $\mathtt{X}_2$ abgelegt und erst danach beim Erzeugen des Abschlusses gleich auch dort eingetragen. Somit spart man nicht nur das Erzeugen eines Variable-Knotens im Speicher, auch eine Unifikation kann durch einen simplen Kopier-Befehl ersetzt werden.

Abbildung 3.9 zeigt den Inhalt der X-Register und des Speichers nach Ausführung dieses Codestücks.

Applikation

Zur Implementierung der Applikation reicht eine Instruktion aus:

`apply` R n
: Die Instruktion prüft zunächst, ob eine Signalbehandlung notwendig ist. Wenn ja wird

der aktuelle Thread gestoppt und die Signalbehandlung durchgeführt. Andernfalls wird gewartet, bis R einen Konstruktor- oder Funktions-Knoten enthält.

Enthält R einen Konstruktor-Knoten c, dann wird ein neues n-stelliges Tupel mit Marke c erzeugt, dessen i-tes Argument gerade der Inhalt von $\mathtt{X}_i$ ist. Das neu erzeugte Tupel wird in $\mathtt{X}_0$ abgelegt.

Enthält R dagegen eine Funktion f mit Arität n, so wird ein neuer *zusätzlicher* Auftrag

$$\langle \mathtt{PC}, \mathbf{nil}, \mathtt{G} \rangle$$

auf dem aktuellen Thread erzeugt, wobei `PC` auf das Segment des Codes des Rumpfes von f verweist, `G` zeigt auf die Werte der freien Variablen von f, der L-Zeiger des Auftrages bleibt leer.

Beispiel Die Parameterübergabe bei der Funktionsapplikation erfolgt wie zuvor bereits erwähnt über die X-Register. Die aufgerufene Funktion liefert ihr Ergebnis im Register $\mathtt{X}_0$ zurück.

Eine Applikation der Form

```
let val x = f(y,z) in ... end
```

wird dann wie folgt umgesetzt:

```
move Ry X0
move Rz X1
apply Rf 2
move  X0 Rx
```

Auf die Instruktion `apply` folgt also noch eine `move` Instruktion, die das Ergebnis des Funktionsaufrufes in das Register rettet, das für `x` vorgesehen ist.

Abbildung 3.10 verdeutlicht das Vorgehen bei der Ausführung einer konkreten Applikation der Form `apply` $\mathtt{X}_2$ `2`. Die Abbildung zeigt die Situation exakt nach Ausführung dieser Instruktion, bevor die erste Instruktion des Rumpfes ausgeführt wurde.

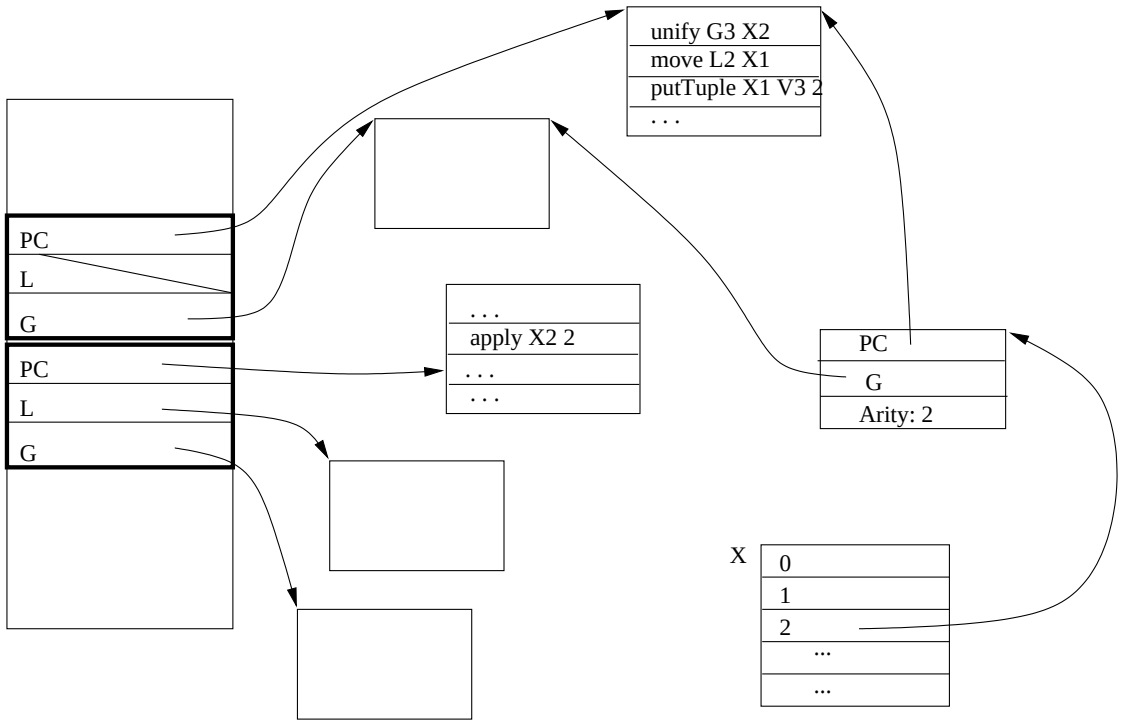
Funktionsrumpf

Zur Implementierung eines Funktionsrumpfes werden neben den bisher vorgestellten Instruktionen, die folgenden neuen Instruktionen verwandt:

`allocate` n
: Alloziert einen neuen (uninitialisierten) L-Registersatz der Größe n. Der L-Zeiger des aktuellen Auftrags zeigt nach Ausführung der Instruktion auf diesen Registersatz, der alte Wert des L-Zeigers geht verloren.

`deallocate`
: Gibt den Registersatz frei, auf den der L-Zeiger des aktuellen Auftrags verweist und setzt den L-Zeiger auf **`nil`**.

Abbildung 3.10 Ausführung einer Applikation erzeugt einen neuen Auftrag



`return` R
: Kopiert den Inhalt des Registers R ins Register $\mathtt{X}_0$. Danach wird der oberste Auftrag des aktuellen Threads T gelöscht und der darunterliegende Auftrag zur Ausführung gebracht. Wenn kein weiterer Auftrag existiert, wird T terminiert und der nächste ausführbare Thread zur Ausführung gebracht.

Der Code für den Rumpf einer Funktion folgt im allgemeinen folgendem Schema:

1. Mittels `allocate` wird zunächst eine hinreichend große Umgebung alloziert, die die lokalen Variablen der Funktion einschließlich der formalen Parameter aufnimmt.
2. Der Aufrufer hat dafür gesorgt, daß die n aktuellen Parameter in den Registern $\mathtt{X}_0$ bis $\mathtt{X}_{n-1}$ befinden. Von dort werden diese nun mittels einer eigenen `move` Instruktion für jeden Parameter in ein L-Register kopiert.
3. Danach folgt der eigentliche Code für den Rumpf der Funktion.
4. Abschließend wird mit `deallocate` die Umgebung wieder freigegeben.
5. Mittels `return` R erfolgt ein Rücksprung zum Aufrufer mit dem Inhalt von R als Ergebnis.

Hierbei handelt es sich nur um ein einfaches, allgemeines Muster. Im konkreten Fall kann der Compiler je nach Struktur des Rumpfes einer Funktion verschiedene davon mitunter stark abweichende Optimierungen vornehmen. Besteht beispielsweise im einfachsten Fall der Rumpf nur aus **`unit`**, so reicht eine einzige Instruktion `return unit` aus.

In anderen Fällen müssen nicht immer alle Parameter in L-Register kopiert werden: das ist etwa dann der Fall, wenn ein Parameter x überhaupt nicht im Rumpf verwandt wird, oder wenn x nur so kurzfristig benötigt wird, daß er in einem X-Register gehalten werden kann (vgl. Abschnitt 5.6).

Auch bei der Allokation der L-Register kann der Übersetzer von obigem Schema abweichen: so kann es beispielsweise günstiger sein, nicht einen großen L-Registersatz für den gesamten Rumpf zu allozieren, sondern in mehreren Schritten verschiedene kleine Sätze anzulegen und wieder freizugeben. Wir werden auf diesen Aspekt noch einmal in Abschnitt 5.6.3 gesondert eingehen.

Beispiel

Als Beispiel wollen wir die Übersetzung der folgenden Funktion betrachten:

```
...
fun f(u,v) =
  let val w = u+v;
      val x = g(u,w)
  in
      x+v
  end
```

Zunächst wird im Segment, in dem diese Anweisung vorkommt, eine `fun` Anweisung eingesetzt, die eine Funktion mit einer globalen Variablen (nämlich `g`) erzeugt:

```
move R_g X0
fun R_f 2 1 S_f
```

Diese referiert das Segment S_f, das den Code für den Rumpf enthält:

```
 1 segment S_f
 2   allocate 3
 3   move X0 L0        % L0: u
 4   move X1 L1        % L1: v
 5   plus L0 L1 L2     % L2: w
 6   move L0 X0        % u --> X0
 7   move L2 X1        % w --> X1
 8   apply G0 2        % x --> X0
 9   plus X0 L1 X1     % x+v --> X1
10   deallocate
11   return X1
12 end
```

Dieser Code ist noch suboptimal; obgleich wir Optimierungen erst in Kapitel 4 diskutieren werden, wollen wir an dieser Stelle kurz auf ein paar Verbesserungen eingehen: beispielsweise kann die Instruktion in Zeile 6 eingespart werden, da `u` sich bereits in X_0 befindet und kein Funktionsaufruf dazwischenliegt, der die Registerinhalte eventuell verändern könnte. Wenn man diese Instruktion streicht, erkennt man, daß L_0 jetzt gar nicht mehr lesend verwandt wird, somit kann

auch Zeile 3 wegfallen. Wodurch wiederum in Zeile 2 nur noch eine Umgebung mit 2 Elementen alloziert werden muß; dann muß man aber auch die Verweise auf $\texttt{L}_1$ und $\texttt{L}_2$ im danach folgenden Code entsprechend anpassen.

3.9 Case

Die Implementierung des Case-Ausdruckes stellt eine besondere Herausforderung dar. Deswegen widmen wir diesem Aspekt einen eigenen Abschnitt.

Bei erster Betrachtung sieht ein Case in Oz zunächst den aus imperativen Sprachen bekannten Verzweigungskonstrukten sehr ähnlich. So scheint die Implementierung von

```
case [x,y] of [1,2] => ... | _ => ...
```

zunächst nicht viel schwieriger, als die Übersetzung etwa eines C-Ausdrucks der Form

```
if (x==1 && y==2) { ... } else { ... }
```

Bei genauerer Betrachtung ergeben sich allerdings deutliche semantisch bedingte Unterschiede. Diese rühren einerseits vom nebenläufigen Charakter der Sprache her; sie sind aber vor allem durch die Tatsache bedingt, daß die Semantik des Case auf Entailment und Disentailment basiert. Somit stellt deren korrekte und effiziente Implementierung eine Herausforderung dar, auf deren Lösung wir in diesem Abschnitt eingehen werden.

Wir werden dabei so vorgehen, daß zunächst im folgenden Abschnitt 3.9.1 ein Überblick über die verschiedenen Aspekte gegeben wird, die im Hinblick auf eine korrekte Implementierung zu beachten sind. Daran anschließend folgt in Abschnitt 3.9.2 eine vertiefende Darstellung der Umsetzung in der Maschine.

3.9.1 Was muß die Implementierung leisten?

Bei einem Case-Ausdruck der Form

```
case x of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

muß, wie in Abschnitt 2.10.5 erläutert, für die Wächter p_i sukzessive entschieden werden, ob diese vom Speicher subsumiert oder dissubsumiert werden. Die Maschine wird dabei so vorgehen, daß versucht wird, x mit p_i zu unifizieren. Schlägt dies fehl, wird zu p_{i+1} übergegangen. Bei erfolgreicher Unifikation muß geprüft werden, ob Variablen aus x gebunden wurden: In diesem Fall muß dann die Auswertung suspendieren und zu einem späteren Zeitpunkt wieder aufgenommen werden.

Bereits an einem einfachen Beispiel

```
let con F;
    con G
in ...
   case x of G(23,F(y)) => ...
   ...
end
```

lassen sich gut verschiedene Aspekte verdeutlichen, die für die Korrektheit der Implementierung zu beachten sind. Man beachte hierbei, daß `y` als einzige Variable im Wächter deklariert wird, da die anderen Variablen ja bereits an Konstruktoren gebunden sind.

Für die Maschine ist es nicht möglich, bei der Auswertung des Wächters diesen in einem atomaren Schritt zum Speicher hinzuzufügen. Vielmehr muß der Speicher *inkrementell* erweitert werden. So wird in obigem Beispiel zunächst geprüft, ob `x` an ein zweistelliges Tupel mit Marke `G` gebunden ist. Danach wird getestet, ob es sich bei dem ersten Argument t_1 von `x` um die Zahl 23 handelt. Handelt es sich bei t_1 um einen VAR-Knoten, so ist nun bereits klar, daß der Wächter nicht subsumiert sein kann. Es wäre aber falsch bereits jetzt mit der Auswertung des Wächters aufzuhören und direkt eine Suspension zu erzeugen, da immer noch Dissubsumption vorliegen kann. So muß auch noch das zweite Argument von `x` mit `F(y)` verglichen werden.

Muß die Auswertung suspendieren, weil weder Subsumption noch Dissubsumption eines Wächters entschieden werden können, so muß sichergestellt werden, daß das *Wecken der Suspension* auch wieder zum richtigen Zeitpunkt erfolgt. Es muß daher einfach sein, die Menge derjenigen Variablen zu bestimmen, bei deren Änderung ein neuerliches Testen auf (Dis)subsumption des Wächters nötig wird.

Hat `x` aus dem Beispiel die Form `G(z,z)`, wobei `z` auf einen VAR-Knoten verweist, dann ist zu beachten, daß es nicht ausreicht, `z` isoliert auf Konsistenz mit `23` und `F(y)` zu testen: jede einzelne Bedingung ist zwar noch erfüllbar, während aber beide Bedingungen nicht gleichzeitig erfüllt werden können, so daß in diesem Fall Dissubsumption vorliegt. Somit muß zur Entscheidung, ob der Wächter im Beispiel erfüllt ist, zunächst `z` an `23` gebunden werden, bevor dann getestet wird, ob `z` mit `F(y)` verträglich ist. Es reicht also kein einfaches *Matching*, sondern es müssen *Unifikationen* durchgeführt werden. Die Implementierung muß also einen Mechanismus zur Verfügung stellen, der es erlaubt, Variablenbindungen temporär im Speicher vorzunehmen und diese später gegebenenfalls wieder zurückzunehmen.

Bei der Entscheidung, ob ein Wächter subsumiert ist, ist es wichtig, eine Unterscheidung in *lokale* Variablen des Wächters (das sind diejenigen Variablen, die im Wächter deklariert wurden) und *globale* (= freie) Variablen des Wächters vorzunehmen. Hat in obigem Beispiel etwa `x` die Form `G(z,F(42))`, wobei `z` einen VAR-Knoten referiert, dann muß die Auswertung suspendieren, da es sich bei `z` um eine globale Variable handelt. Wird dann später `z` an die Zahl 23 gebunden, dann ist der Wächter subsumiert, da es sich bei `y` ja um eine lokale Variable handelt und demnach die Gleichung $\exists$`y`$=$`G(23,F(y))` logisch impliziert wird.

Zum Entscheiden von Subsumption (nicht für Dissubsumption) muß prinzipiell also für jede Variable erkennbar sein, wo ihr Quantor sitzt, oder genauer: ob sie lokal oder global ist. Diese Eigenschaft einer Variable kann sich allerdings *dynamisch ändern*. Dann nämlich wenn der Wächter subsumiert ist; dadurch werden alle lokalen Variablen des Wächters zu globalen Variablen.

Die Unterscheidung in lokale und globale Variablen stellt für die Implementierung des tiefen Berechnungsmodells eine deutliche Verkomplizierung dar, insbesondere da hier Berechnungsräume beliebig geschachtelt sein können, aber auch wieder verschmolzen werden können [Sch99].

Wir werden allerdings im folgenden Abschnitt sehen, daß für das in dieser Arbeit diskutierte flache Modell die Unterscheidung in lokale und globale Variablen wesentlicher einfacher realisiert werden kann. Dies erlaubt es, Case deutlich einfacher und effizienter zu implementieren als im tiefen Modell, wo im allgemeinen lokale Berechnungsräume aufgebaut werden müssen. Allerdings können die hier vorgestellten Techniken zur Implementierung von sogenannten *flachen* Wächtern direkt auch beim tiefen Oz angewandt werden: lediglich für wirklich tiefe Wächter (die in der Praxis sehr selten auftreten) müssen lokale Berechnungsräume aufgebaut werden. So kann die hier vorgestellte Implementierungstechnik auch auf das Gros der Konditionale im tiefen Oz angewandt werden.

3.9.2 Prinzipielles Vorgehen

Wie weiter oben beschrieben wird Subsumption eines Wächters p_i so entschieden, daß x testweise mit p_i unifiziert wird. Im Falle von Suspension oder Dissubsumption muß dann aber der Speicher wieder in seinen Originalzustand zurückversetzt werden. Wie wir im folgenden sehen werden, bestehen die einzigen Änderungen am Speicher, die wieder zurückgenommen werden müssen im Binden von Variablen. Hierzu verwendet die Maschine ein spezielles Register `trail` auf einen Keller, das *Spur* genannt wird: die ursprünglichen Werte der Variablen, die beim Ausführen des Wächters gebunden werden, werden darauf gesichert, so daß der Speicher im Falle von Suspension oder Dissubsumption des Wächters in den Ursprungszustand zurückversetzt werden kann. In einer sequentiellen Implementierung ist dieses Vorgehen problemlos möglich. In einer parallelen Implementierung kommen hier andere Techniken zum Einsatz, indem die Variablenbindungen nicht global vorgenommen werden, sondern etwa lokal zu den einzelnen parallelen Ausführungseinheiten in sogenannten Bindungsfeldern gespeichert werden [GJ90, MA95].

Die Entscheidung von Dissubsumption eines Wächters ist einfach: schlägt während der Ausführung des Wächters eine Unifikation fehl, so liegt Dissubsumption vor.

Die Entscheidung von Subsumption ist komplizierter. Hierzu gibt es in der Literatur [ST94] ein Verfahren, das hier angepaßt werden kann: die zu einem Wächter p_i mit lokalen Variablen $\overline{y}$ gehörende Gleichung $x = p_i$ ist relativ zu einem Constraint C subsumiert, genau dann wenn φ' existiert mit

$$C \wedge \exists\overline{y} x = p_i \quad \equiv \quad C \wedge \exists\overline{y}\varphi \quad \equiv \quad C$$

Dabei ist φ eine Konjunktion von Gleichungen der Form $y_i = t_i$ und alle y_i sind lokale Variablen. Nach der Unifikation von x mit p_i läßt sich prinzipiell leicht aus dem Inhalt der Spur ein Gleichungssystem $y_i = t_i$ konstruieren. Subsumption liegt genau dann vor, wenn für jede dieser Gleichungen entweder y_i oder t_i eine lokale Variable ist. Der Test auf Subsumption läßt sich aber noch deutlich vereinfachen, wenn man die Bindung von lokalen Variablen nicht auf der Spur vermerkt. Dieses Vorgehen ist korrekt: da die lokalen Variablen eines Wächters nach außen nicht sichtbar sind, muß somit deren Bindung nach Ausführung des Wächters auch nicht notwendig wieder zurückgenommen werden. Wenn man zudem darauf achtet, daß bei der Unifikation zweier Variablen lokale stets präferiert an globale Variablen gebunden werden, reduziert sich der Test auf Subsumption damit deutlich: nach Ausführung des Wächters liegt genau dann Subsumption vor, wenn die Spur leer ist.

Liegt weder Subsumption noch Dissubsumption vor, so enthält die Spur gerade diejenigen Variablen, auf die suspendiert werden muß: für den aktuellen Thread wird eine Suspension erzeugt, die in die Suspensionsliste *aller* dieser Variablen eingetragen wird. Dabei wird der oberste Auftrag des Threads so gewählt, daß nach dessen Wecken der komplette Code des Wächters noch einmal vollständig ausgeführt wird.

Die Unterscheidung von Variablen in lokal und global geschieht wie folgt: die Instruktionen, die für den Wächter erzeugt werden, arbeiten so, daß (a) alle im Wächter erzeugten Variablen in einer speziellen Liste `locals` aufgesammelt werden und (b) beim Binden einer Variable zusätzlich geprüft wird, ob diese im Register `locals` vorkommt; wenn nicht wird ein entsprechender Eintrag auf der Spur vorgenommen.

Zunächst erscheint der Weg zur Unterscheidung lokal/global über eine explizite Liste `locals` nicht sehr effizient. Wir werden allerdings im Implementierungskapitel in Abschnitt 7.6 aufzeigen, daß dies in der Praxis dennoch sehr effizient realisierbar ist. Wir wollen hier zwar noch nicht vorgreifen, aber an dieser Stelle dennoch so viel verraten: der Test auf Lokalität einer Variable läßt sich sehr billig über einen Adressvergleich im Speicher realisieren, wenn man einfach vor Betreten eines Wächters sich den aktuellen Wert des Haldenzeigers merkt. Eine Variable ist genau dann lokal, wenn sie jünger als dieser gemerkte Wert ist.

Zusammenfassend noch einmal die wichtigsten Schritte bei der Implementierung des Case:

- Im Wächter erzeugte (lokale) Variablen werden in `locals` gemerkt.
- Wird von einem Wächter eine Variable x gebunden, die nicht in `locals` vorkommt, so wird x auf der Spur `trail` gemerkt. Lokale Variablen werden an globale gebunden, aber nicht umgekehrt.
- Schlägt eine Unifikation in einem Wächter fehl, werden alle Bindungen gemäß `trail` zurückgenommen, und die Ausführung mit dem nächsten Wächter fortgesetzt.
- Wird ein Wächter ohne fehlzuschlagen komplett ausgeführt, wird Subsumption entschieden: ist `trail` leer, wird mit der Ausführung des entsprechenden Rumpfes fortgefahren. Andernfalls suspendiert der aktuelle Thread auf alle Variablen, die auf `trail` gemerkt wurden.
- Bei Suspension wird der Code des suspendierten Wächters von vorne noch einmal ausgeführt.

3.9.3 Instruktionen

Der Maschinencode für ein Case der Form

```
case x of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

folgt folgendem allgemeinen Muster:

```
  guardStart L2
  ⟨Code für x = p1⟩
  guardEnd
  ⟨Code für e1⟩
  branch LEnd
L2:
  guardStart L3
  ⟨Code für x = p2⟩
  guardEnd
  ⟨Code für e2⟩
  branch LEnd
L3:
  ...
Ln:
  guardStart LEnd
  ⟨Code für x = pn⟩
  guardEnd
  ⟨Code für en⟩
LEnd: ...
```

Für jeden Wächter p_i wird Code für eine Gleichung der Form $x = p_i$ erzeugt. Dieser Code wird durch eine spezielle Instruktion `guardStart` eingeleitet, die als Argument eine Marke verwendet. Diese Marke verweist auf den Code, der für den nachfolgenden Wächter p_{i+1} erzeugt wurde. Dieser Code wird angesprungen, sobald eine Unifikation im Wächter fehlschlägt. Das Ende eines Wächters wird durch die Instruktion `guardEnd` markiert. Direkt auf diese Instruktion folgt der Code für e_i an dessen Ende ein Sprungbefehl `branch` eingefügt wird, der den Code für den auf das Case folgenden Anweisungen anspringt.

Die Maschine verwendet ein Register `caseStart`. Dieses verweist auf den Anfang des Codes des aktuellen Wächters, also die Instruktion `guardStart` und erfüllt zwei Funktionen: für den Fall des Fehlschlagens einer Unifikation kann die Maschine über `caseStart` erkennen, wo der Code für den nächsten Wächter zu finden ist. Weiter wird `caseStart` für den Fall der Suspension verwendet: in diesem Fall wird ja der Wächter nach dem Wecken noch einmal von vorne ausgeführt, so daß eine Referenz auf den Anfang des Codes benötigt wird.

Während für die Rümpfe e_i kein spezieller Code erzeugt werden muß, werden für die Gleichungen der Wächter einige neue zusätzliche Instruktionen eingeführt.

In Kapitel 2 haben wir festgelegt, daß ein Laufzeitfehler ausgelöst wird, wenn kein Pattern erfüllt ist. In der Implementierung läßt sich dies einfach erreichen, indem der Compiler falls nötig am Ende eines Case noch

```
| _ => error(...)
```

einfügt, so daß hierfür keine speziellen Instruktionen benötigt werden.

Die für Case zusätzlich benötigten Instruktionen im einzelnen:

`branch` L
: Unbedingter Sprungbefehl, der die Instruktionen, die an der Marke L beginnen, zur Ausführung bringt.

`guardStart` L
: Die Instruktion markiert den Beginn eines Wächters. Die Marke L verweist auf den Beginn des Codes des nächsten Wächters. Die Instruktion setzt den Zeiger `caseStart` der Maschine auf die aktuelle Instruktion.

`guardEnd`
: Die Instruktion markiert das Ende eines Wächters: es wird geprüft, ob die Spur `trail` leer ist. Wenn ja, wird die nächste Instruktion ausgeführt. Wenn die Spur nicht leer ist, wird eine Suspension erzeugt und in die Suspensionsliste aller Variablen x, die die Spur enthält, eingetragen. Der aktuelle Auftrag wird so modifiziert, daß er auf `guardStart` verweist, so daß der suspendierte Wächter nach dem Wecken von neuem ausgeführt wird.

`getInt` R V
: Die Instruktion erwartet in V eine Zahl n. Enthält R einen VAR-Knoten, dann wird diese Variable an n gebunden und gegebenenfalls auf der Spur gemerkt. Enthält R die Zahl n, wird die nächste Instruktion ausgeführt. Enthält R schließlich einen Knoten, der keine Variable ist und zudem verschieden von n ist, dann wird failure ausgelöst (also wie zuvor beschrieben die Spur geleert und zum Code des nächsten Wächters gesprungen).

`getCon` R V
: Die Instruktion arbeitet analog zu `getInt`, und erwartet im Unterschied dazu in V einen Konstruktor.

`moveArg` R n R'
: Die Instruktion erwartet in R eine direkte Referenz auf ein Tupel. Sie lädt das n-te Argument dieses Tupels nach R'. „direkte Referenz“ bedeutet hier, daß R nicht auf einen REF-Knoten verweisen darf, da die Instruktion keinen Dereferenzierungsschritt durchführt.

`setArg` R n R'
: Die Instruktion erwartet in R eine direkte Referenz auf ein Tupel und setzt das n-te Argument dieses Tupels auf den Knoten, auf den R' verweist.

`getTuple` R R' n L
: Die Instruktion erwartet in R' einen Konstruktor-Knoten. Zunächst wird der Inhalt von R geprüft: enthält R eine Variable, dann wird ein neues Tupel mit n uninitialisierten Argumenten und mit Marke R' erzeugt und mit R unifiziert. In R wird eine Referenz auf dieses neu erzeugte Tupel abgelegt. Anschließend wird zur Marke L gesprungen.
Enthält R keine Variable, wird geprüft, ob R ein Tupel t mit Marke R' und Breite n enthält, wenn nicht, wird failure ausgelöst (also wie zuvor beschrieben die Spur geleert und zur Marke für **`else`** gesprungen); wenn ja, wird in R eine Referenz auf t abgelegt und die nächste Instruktion ausgeführt.

Wir wollen uns nun der Beschreibung der Codeerzeugung für die Wächter zuwenden. Bei der Codeerzeugung für eine Gleichung $x = p_i$ eines Wächters transformiert der Compiler die Gleichung durch Einführung von lokalen Hilfsvariablen zunächst in eine Folge von Gleichungen der Form $y = t$, wobei es sich bei t entweder um eine Variable, eine Zahl oder ein Tupel mit Variablen als Argumenten handelt. So wird etwa

```
x = F(23,G(y),y)
```

in folgende Form transformiert

```
x = F(u,v,y)
u = 23
v = G(y)
```

Im folgenden beschreiben wir den Maschinencode, der den verschiedenen Varianten von Gleichungen entspricht.

$x = n$, $x = c$
: Handelt es sich bei der rechten Seite einer Gleichung um eine Zahl oder einen Konstruktor, so wird

  `getInt` R_x `n` `getCon` R_x c

  erzeugt.

$x = y$
: Hier muß eine Unifikation durchgeführt werden:

  `unify` R_x R_y

$x = y(z_1, \ldots, z_n)$
: Diesen Fall könnte man so behandeln, daß man zunächst ein neues Tupel mit Marke y und Argumenten z_i erzeugt und dieses mit x unifiziert. Das ist aber nur dann sinnvoll, wenn x auf einen VAR-Knoten verweist. Da x in der Regel zur Laufzeit aber bereits auf ein Tupel verweisen wird, würde dadurch unnötig Speicher und Laufzeit verschenkt.

  Stattdessen erzeugen wir zwei Codeströme C_1 und C_2, die jeden der beiden Fälle abdecken. Diesen Instruktionen voraus wird die Instruktion `getTuple` gestellt. `getTuple` prüft zunächst, ob x an ein Tupel gebunden ist. Wenn ja wird zu C_1 verzweigt, wenn nein wird zunächst ein neues Tupel im Speicher aufgebaut und anschließend zu C_2 gesprungen.

  Wir wollen das Vorgehen an einem kleinen Beispiel erläutern. Wir betrachten folgende Gleichung

  $$x = y(u, v)$$

  und nehmen an, daß es sich hier lediglich bei v um das erste Auftreten einer lokalen Variablen handelt, während u bereits vorher vorkam. Der Maschinencode für diese Gleichung sieht dann so aus:

  `getTuple` R_x R_y n `L1`
  `moveArg` R_x `0` R_{aux}
  `unify` R_u R_{aux}

```
    moveArg R_x 1 R_v
    branch L2
  L1:
    setArg R_x 0 R_u
    newVar R_v
    setArg R_x 1 R_v
  L2:
    ...
```

Die Ausführung von `getTuple` erzeugt ein neues Tupel und springt zur Marke `L1`, falls x auf einen Variable-Knoten verweist. Der Code an der Marke `L1` übernimmt die Behandlung der Argumente u und v: mittels `setArg` wird das erste Argument des gerade erzeugten Tupels (mit noch uninitialisierten Argumenten) auf den Wert von u gesetzt. Da v eine lokale Variable ist, die an dieser Stelle zum ersten Mal auftritt, wird zunächst ein neuer VAR-Knoten erzeugt, auf den dann das zweite Argument des Tupels verweist.

Findet `getTuple` allerdings bereits ein Tupel vor, so werden die direkt darauf folgenden Instruktionen ausgeführt. Da u bereits auf einen Knoten im Speicher verweist, wird hier mittels `moveArg` das erste Argument von x in ein Hilfsregister R_{aux} geladen und dann mittels `unify` eine Unifikation mit u durchgeführt. Anders ist die Situation bei v, da wir angenommen haben, daß v zum ersten Mal an dieser Stelle auftritt: es reicht hier mittels `moveArg` eine Referenz auf das zweite Argument von x in das Register von v zu laden.

3.9.4 Beispiel

Als Beispiel wollen wird den Maschinencode von folgendem Ausdruck betrachten:

```
case x of                          guardStart L3
  F(u,u,4711) => e_1               getTuple R_x F 3 L1
  y           => e_2               moveArg  R_x 0 R_u
                                   moveArg  R_x 1 X_10
                                   unify    X_10 R_u
                                   moveArg  R_x 2 X_10
                                   getInt   X_10 4711
                                   branch L2
                               L1: newVar R_u
                                   setArg R_x 0 R_u
                                   setArg R_x 1 R_u
                                   setArg R_x 2 4711
                               L2: guardEnd
                                   ⟨ Code für e_1⟩
                                   branch L4
                               L3: ⟨ Code für e_2⟩
                               L4: ...
```

Der erste Wächter muß wie zuvor beschrieben durch zwei Codeströme aufgelöst werden, die von `getTuple` angesteuert werden. Man beachte, daß das erste Vorkommen von `u` jeweils unterschiedlich zum zweiten behandelt wird: so wird im Lese-Strom zunächst das entsprechende Argument von `x` im Hilfsregister X_{10} gerettet und dann mit dem zweiten Argument unifiziert, während im Schreib-Strom eine neue Variable erzeugt wird. Die beiden Codeströme werden bei `L2` wieder zusammengeführt. Auf die Instruktion `guardEnd` folgt der Code für den ersten Rumpf, während

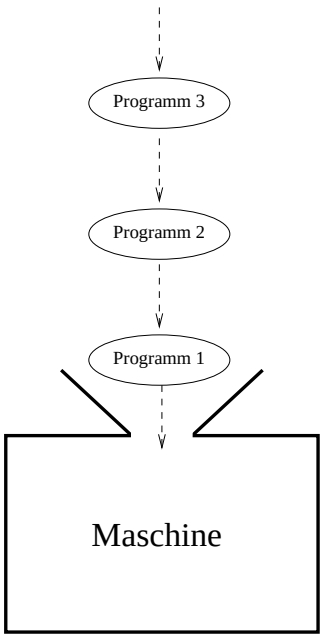
bei `L3` der Else-Fall steht: dadurch, daß die zweite Klausel immer erfüllt ist, kann hier das Einsetzen von `guardStart`, das direkt von `guardEnd` gefolgt wird, eingespart werden, so daß hier direkt der Code von e_2 abgelegt werden kann. Bei `L4` beginnen die Instruktionen für die Anweisungen, die auf das Case folgen.

3.10 Inkrementalität und getrennte Übersetzung

Zu Anfang des Kapitels haben wir mit einem einfachen Modell des Zusammenspiels zwischen Maschine und Übersetzer begonnen. Wir sind dabei davon ausgegangen, daß der Compiler jeweils ein in sich geschlossenes Programm übersetzt und die Maschine lediglich in der Lage ist, genau ein Programm auszuführen und nach dessen Bearbeitung direkt wieder terminiert.

In diesem Abschnitt werden wir sehen, daß sich dieses einfache Modell sehr leicht auf das interaktive Modell, das der Mozart-Programmierschnittstelle OPI [MMP+97] zugrunde liegt, erweitern läßt. Wir werden im folgenden so vorgehen, daß wir zunächst beschreiben, wie die Maschine erweitert werden muß, damit sie mehrere Programme direkt nacheinander ausführen kann, so wie es in Abbildung 3.11 dargestellt ist. Danach werden wir zeigen, wie Information aus einem Programm in nachfolgende Programme fließen kann.

Abbildung 3.11 Interaktivität: Maschine führt mehrere Programme nacheinander aus.



3.10.1 Ausführung mehrerer Programme

Die Erweiterung der Maschine dahingehend, daß sie mehrere Programme nacheinander ausführen kann, ist denkbar einfach: wir müssen nur die Funktionalität des Laders in der Sprache verfügbar machen. Wir wollen das Vorgehen hier kurz skizzieren: wir gehen davon aus, daß das System eine vordefinierte einstellige Funktion

```
loadProgram(IO)
```

zur Verfügung stellt. `IO` ist dabei ein Eingabekanal der Maschine, auf dessen genaue Beschaffenheit wir hier nicht weiter eingehen wollen. Die Funktion `loadProgram` prüft nun, ob an diesem Kanal ein neues Programm verfügbar ist und ruft dann den Lader auf, der wie in Abschnitt 3.5.4 beschrieben dieses Programm in ein ausführbares Format transformiert. Allerdings bringt er es nun nicht direkt zur Ausführung, sondern erzeugt eine neue nullstellige Funktion `f` daraus, die `loadProgram` nun als Ergebnis zurückliefert.

Damit ist die Erweiterung der Maschine bereits abgeschlossen: die Idee ist nun, daß die Maschine nach wie vor beim Booten ein bestimmtes Programm lädt und dieses ausführt. Ein solches Programm kann nun allerdings auch wie folgt skizziert werden:

```
fun nextProgram(IO) =
   let val f = loadProgram(IO)
   in
     f();
     nextProgram(IO)
   end
```

Die Funktion `nextProgram` lädt in einer Schleife ein Programm in Form einer Funktion `f` nach dem anderen und bringt es durch Applikation zur Ausführung. Dabei wird zwar erst dann das nächste Programm geladen, wenn `f` bearbeitet wurde, allerdings kann der Rumpf von `f` ja beliebige neue Threads erzeugen, so daß bereits das nächste Programm geladen werden kann, auch wenn `f` noch nicht vollständig bearbeitet wurde.

Die Maschine lädt nun so lange Programme, bis die beiden folgenden Bedingungen erfüllt sind:

1. Alle Ein-/Ausgabe-Kanäle sind geschlossen.
2. Alle Threads sind entweder terminiert oder suspendiert.

Sind diese beiden Bedingungen erfüllt, so gibt es keine Möglichkeit mehr, daß noch ein neuer Thread erzeugt wird oder ein Thread in den Zustand ausführbar übergeht. Somit kann die Maschine terminieren, wenn beide Bedingungen erfüllt sind.

3.10.2 Export von Variablen

Mit dem zuvor beschriebenen Mechanismus ist es möglich, mehrere Programme nacheinander auszuführen, dies ist jedoch aus praktischer Sicht noch nicht ausreichend, da es noch nicht möglich ist, daß ein Programm auf die Variablen vorangehender Programme zugreift. Hierzu ist ein neues Sprachkonstrukt notwendig

declare $x = e_1$ **in** e_2

Semantisch gesehen ist diese Konstrukt äquivalent zu

let val $x = e_1$ **in** e_2 **end**

Ein wichtiger Unterschied besteht allerdings darin, daß **declare** einen offenen Skopus hat: so ist x nicht nur in e_2 sichtbar, sondern auch in allen nachfolgenden Programmen.

So kann beispielsweise ein Programm eine Funktion `f` wie folgt exportieren:

```
declare f = fn(x,y) => x+y
in unit
```

Ein darauf folgendes Programm kann nun neue Variablen exportieren und kann `f` dazu verwenden:

```
declare g = fn(x) => f(x,x)
in unit
```

Insbesondere ist es auch möglich, daß ein drittes Programm nun `f` neu deklariert (dieses Mal als Zahl)

```
declare f=5 in unit
```

so daß nun jeder direkte Zugriff auf das erste `f` verloren ist. Nichtsdestotrotz bedingt nach wie vor jeder Aufruf von `g` selbstverständlich einen Aufruf des ursprünglichen `f`.

3.10.3 Die Exporttabelle

Zum Zweck des Imports und Exports von Variablen verwaltet die Maschine eine *Exporttabelle*. Das ist eine Tabelle, die eine Zuordnung von Zeichenketten (den exportierten Variablen) zu Knoten im Speicher vornimmt. Der Zugriff auf diese Tabelle wird ausschließlich über zwei spezielle Preludeinstruktionen ermöglicht:

`export` $\mathtt{V}_i$ c
: Trägt in die Exporttabelle unter der Zeichenkette c den Knoten ein, auf den das V-Register $\mathtt{V}_i$ verweist. Wenn bereits ein Eintrag für c existiert, wird dieser überschrieben.

`import` $\mathtt{V}_i$ c
: Lädt den Knoten, der unter der Zeichenkette c in der Exporttabelle gespeichert ist, in V-Register $\mathtt{V}_i$. Wenn kein Eintrag für c gefunden wird, erfolgt eine Fehlermeldung.

Wir wollen das Vorgehen an einem kleinen Beispiel verdeutlichen. Das folgende Programm

```
declare
  x=13;
  y=lvar()
in
  z(y)
```

exportiert die Variablen `x` und `y` und importiert `z`. Das Maschinenprogramm dazu, sieht dann so aus:

```
prelude
  vnewInt V0 13
  export V0 "x"
  vnewVar V1
  export V1 "y"
  import V2 "z"
end
segment S0
  move V1 X0
  apply V2 1
  return X0
end
```

Hier kann die Zahl 13 zunächst direkt über V_0 unter dem Namen `"x"` exportiert werden, während für `y` vor dem Export zunächst eine Variable im Speicher erzeugt werden muß (die aber vielleicht durch den Aufruf von `z` gebunden wird).

Man beachte, daß das vergleichsweise teure Nachschlagen in der Exporttabelle bereits zur Ladezeit des Codes erfolgt. Alle Zugriffe zur Programmlaufzeit geschehen dagegen effizient über V-Register.

3.10.4 Optimierungen über Programmgrenzen

Wir wir in Kapitel 4 sehen werden, kann der Compiler, wenn bei der Übersetzung eines Programmes P die Werte von importierten Variablen bekannt sind, davon abhängig Optimierungen vornehmen. Weiß der Compiler beispielsweise, daß `x` den Wert `13` hat und kommt in P ein Case der Form

```
case x of
  13 => f(y)
| _  => g(y)
```

vor, so kann dieses zu

```
f(y)
```

vereinfacht werden. Diese Optimierung ist dann besonders wirkungsvoll, wenn sich das Case innerhalb einer häufig aufgerufenen Funktion befindet.

Nun ist aber nicht sicher, ob ein Programm das unter diesen Prämissen übersetzt wurde, später auch in eine Maschine geladen werden wird, die in ihrer Exporttabelle unter der Variable `"x"` eine Referenz auf die Zahl `13` enthält.

Will man Optimierungen über Programmgrenzen hinweg nicht völlig verbieten, so wäre eine naheliegende Möglichkeit, die `import` Instruktion so zu erweitern, daß sie zusätzlich noch prüft, ob der Wert der importierten Variable gewisse (mitunter wohl recht komplexe) Bedingungen erfüllt. Diese Erweiterung ist allerdings gar nicht nötig, da das Format für Maschinenprogramme bereits leistungsfähig genug ist: der Compiler kann nämlich zu Beginn des ersten Segments (das ja direkt nach dem Laden ausgeführt wird) noch den Code für ein Case der Form

```
case x of
  13 => unit
| _  => error(´Assertion failed: x must be 13´)
end
```

ablegen. So wird sichergestellt, daß vor Ausführung eines Programmes P auch komplexe Bedingungen, die bei der Übersetzung von P berücksichtigt wurden, auf Korrektheit geprüft werden können.

Das Format, in dem Maschinenprogramme vom Compiler abgelegt werden, ist sehr leistungsfähig und ausbaubar. Wir wollen dies im folgenden an zwei Beispielen belegen. Die Aspekte von Persistenz und Verteilung sind zwar nicht Gegenstand dieser Arbeit, wir wollen an dieser Stelle aber dennoch kurz skizzieren wie das Format von Maschinenprogrammen für diese Bereiche genutzt werden kann.

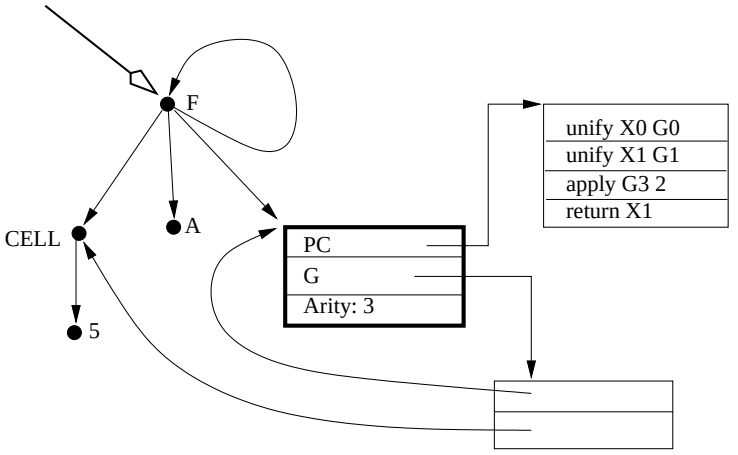
3.11 Persistenz

Unter dem Begriff *Persistenz* wollen wir die Möglichkeit verstehen, Daten auf langlebige Speichermedien auszulagern, so daß beispielsweise die Möglichkeit besteht, Werte, die in einer Sitzung von einem Programm erzeugt wurden, in einer späteren Sitzung (womöglich an einem ganz anderen Ort unter einem anderen Betriebssystem) wieder einzulesen und weiter zu verarbeiten.

Die Idee einen bestimmten Wert w persistent zu machen, besteht nun ganz einfach darin, ein Programm zu generieren, dessen Ausführung gerade w erzeugt. Dazu muß das Format der Maschinenprogramme noch nicht einmal modifiziert werden: in der Form, in der wir es vorgestellt haben, ist es bereits leistungsfähig genug, um neben Zahlen, Konstruktoren und Tupeln auch Zellen und Funktionen zu speichern.

Wir wollen das Vorgehen an einem konkreten Beispiel verdeutlichen: Abbildung 3.12 zeigt links ein vierstelliges Tupel und rechts den zugehörigen Code zum Aufbau des Tupels. Der Code liefert das Tupel als Ergebnis in $\mathtt{X}_0$ zurück (dies könnte alternativ auch zum Beispiel über $\mathtt{V}_0$ oder die Exporttabelle geschehen). Beim Maschinenprogramm haben wir diesmal der Übersichtlichkeit halber die Prelude weggelassen.

Abbildung 3.12 Ein Tupel und der Code, der es persistent macht




```
segment S0
  move 5 X2
  newRef X1 X2
  fun X0 3 2 S1
  move X0 X2
  newTuple X0 F 4
  setArg X0 0 X1
  setArg X0 1 A
  setArg X0 2 X2
  setArg X0 3 X0
end
segment S1
  unify X0 G0
  unify X1 G1
  apply G3 2
  return X1
end
```

Segment S_0 enthält den eigentlichen Code zum Aufbau des Tupels, während S_1 den Rumpf der Funktion enthält, die über das dritte Argument des Tupels erreichbar ist. Diese Funktion hat zwei globale Variablen, das ist einmal die Funktion selbst und zum anderen eine Zelle (mit Inhalt `5`), die auch über das erste Argument des Tupels erreichbar ist.

Die ersten beiden Instruktionen in S_0 dienen zum Aufbau der Zelle samt Inhalt in X_1. Danach kann bereits die Funktion mittels `fun` in X_0 erzeugt werden. Direkt darauf wird dann mittels `move` das Register X_0 frei gemacht, da hierhin das Tupel kommt. Nun müssen nur noch mittels `setArg` die einzelnen Argumente geschrieben werden. Man beachte dabei, daß auch die Erzeugung eines Zyklus (im letzten Argument des Tupels) keine besondere Hürde darstellt, wenn man wie hier von außen nach innen vorgeht und bei einem neuerlichen Auftreten eines Wertes einfach das Register benennt, in dem das erste Auftreten festgehalten wurde. Außerdem ist zu beachten, daß der Code der Funktion keine anderen Segmente referiert; andernfalls müssen diese selbstverständlich auch mit ausgegeben werden.

Somit lassen sich alle Werte erzeugen. Man kann aber auch noch weitergehen und zum Beispiel Threads persistent machen: hierzu muß allerdings das Format von Maschinenprogrammen erweitert werden, indem man neue Preludeinstruktion oder zusätzliche Maschineninstruktionen definiert. Schließlich kann man auch letztlich alle Einschränkungen aufheben und durch Hinzunahme hinreichend vieler Instruktionen erlauben, den gesamten Inhalt des Speichers zu sichern, so daß man beispielsweise am Ende eines Tages seine Arbeit sichern und am nächsten Tag (vielleicht in einer anderen Umgebung) an der genau gleichen Stelle wieder aufnehmen kann. Wobei in diesem Fall allerdings noch die Frage zu lösen ist, wie offene Verbindungen zu anderen Prozessen gesichert und wieder restauriert werden können.

3.12 Verteilung

Unter dem Begriff *Verteilung* verstehen wir die Möglichkeit, daß mehrere Oz Maschinen, die in der Regel auf verschiedenen unter Umständen weit entfernten Rechnern laufen, miteinander

Kontakt aufnehmen und Daten (insbesondere auch Funktionen) austauschen können, um so beispielsweise eine größere Aufgabe gemeinsam zu lösen. Wir wollen an dieser Stelle nur kurz den Aspekt des Austauschs von zustandsfreien Werten beleuchten und nicht tiefer in diese Thematik einsteigen und verweisen den interessierten Leser auf [VHB+97], [HVS97] und [HVBS98].

Unter zustandsfreien Werten verstehen wir alle Werte ohne Zellen und Variablen. Der Austausch zustandsfreier Werte kann prinzipiell bereits mit den Techniken der Persistenzmachung aus dem vorangehenden Abschnitt bewerkstelligt werden. So sichert eine Lokation L einen Wert in eine Datei, die dann zur anderen Lokation L' mit konventionellen Medien und Protokollen (z.B. FTP, HTTP, etc.) übertragen wird. Das Problem bei diesem Vorgehen ist allerdings, daß diese Operation die Gleichheit von Werten nicht erhält. Sie funktioniert nur für Tupel und Zahlen korrekt, weil auf diesen Werten die Gleichheit über deren Struktur definiert ist. Wird hingegen eine Funktion oder ein Konstruktor von L nach L' übertragen, so wird jedesmal bei L' eine *Kopie* davon angelegt. Wird also beispielsweise im Laufe einer komplexen Berechnung die Funktion f zweimal von L nach L' übertragen, so würde bei L' jedesmal eine neue Funktion erzeugt, so daß beide verschieden sind. Umgekehrt würde auch ein Übertragen von f zu L' und dann wieder von dort zurück zu L bei L zur Erzeugung zweier verschiedener Funktionen führen.

Zur Lösung dieses Problems werden *globale Namen* verwandt; das sind Zeichenketten, die weltweit eindeutig sind. Abstrakt kann man sich das Vorgehen so vorstellen, daß es eine einzige zentrale Lokation ZL gibt, die auf Anforderung eindeutige Namen vergibt.

Jede Maschine verwaltet somit zusätzlich eine *Namen-Tabelle*, die solche globalen Variablen den entsprechenden Funktionen beziehungsweise Konstruktoren zuordnet. Wird nun beispielsweise ein Konstruktor c verschickt, so wird zunächst geprüft, ob die Tabelle bereits einen Namen B für c enthält; wenn nicht wird ein neuer erzeugt und in die Tabelle eingetragen. Wurde bisher beim Persistentmachen eine Instruktion der Form

```
newCon X5
```

ausgegeben, so wird diese ersetzt durch eine neue Instruktion

```
newGlobalCon X5 B
```

die bei Ausführung durch den Empfänger folgendes bewirkt: es wird zunächst in der Namenstabelle nach einem Eintrag für B gesucht. Wird einer gefunden, so wird in `X5` eine Referenz auf den Knoten im Speicher abgelegt, auf den der Tabelleneintrag verweist. Wird kein Eintrag gefunden, so wird ein neuer Konstruktor-Knoten im Speicher erzeugt, dieser unter dem Namen B in der Namenstabelle gespeichert und ebenfalls eine Referenz darauf nach `X5` geschrieben.

Dieses Vorgehen funktioniert auch dann noch korrekt, wenn man den Konstruktor nun auch noch an eine dritte Lokation weiterreicht.

Für die Instruktion `fun` wird durch Einführung einer neuen Instruktion `globalFun` analog verfahren.

Aus technischer Sicht es es nun aber ungünstig die Verwaltung der globalen Namen von einer zentralen Stelle ZL vornehmen zu lassen. Stattdessen kann man auch jeder Lokation einen unendlichen Ausschnitt aus dem globalen Namensraum zuweisen, den sie selbst in eigener Regie ohne Interaktion mit anderen Lokationen verwalten kann. Wichtige Forderungen an die Wahl der globalen Namen sind die, daß es für jede Lokation möglich sein muß, (a) schnell und (b) beliebig viele solcher Namen zu erzeugen. In der Praxis hat sich folgendes Vorgehen bei der Erzeugung neuer globaler Variablen bewährt: die globalen Variablen setzen sich zusammen aus (1)

der (Internet-)Adresse der Lokation, (2) dem Startzeitpunkt der Oz Maschine auf dieser Lokation und (3) dem Wert eines Zählers, der bei jeder Erzeugung einer neuen globalen Variablen erhöht wird. Dies erfüllt die beiden Kriterien (a) und (b).

Dieses Beispiel stellt noch einmal die Leistungsfähigkeit des Formats für Maschinenprogramme unter Beweis: durch die Hinzunahme zweier Instruktionen wird es über dieses Format möglich, zustandsfreie Werte mit anderen Lokationen auszutauschen.

Zusammenfassung

- Die Implementierung von L wird durch Übersetzung in Code für eine registerbasierte *virtuelle Maschine* bewerkstelligt.
- *Gebundene Variablen* werden durch spezielle `REF`-Zellen im Speicher dargestellt.
- Die Variablen der Quellsprache werden über Register adressiert. Die Maschine kennt vier Adressierungsarten: *X-Register* für temporäre Werte, *L-Register* und *G-Register* für die lokalen respektive freien Variablen einer Funktion und *V-Register* für konstante Werte.
- Ein *Maschinenprogramm* besteht aus einer *Prelude* und einer Folge von *Segmenten*. Ein Segment enthält den Maschinecode je eines Thread- oder Funktionsrumpfes.
- Ein Thread ist ein Stapel von *Aufträgen*. Jeder Auftrag entspricht einem Abschluß, der als Tripel der Form $\langle$`PC`, `L`, `G`$\rangle$ dargestellt ist.
- Bei jeder Funktionsapplikation wird auf die Notwendigkeit einer *Signalbehandlung* getestet. Dies ist ausreichend, um *Fairneß* zu garantieren.
- Jede Variable verwaltet ihre eigene *Suspensionsliste*, die über eine Indirektion diejenigen Threads enthält, die auf diese Variable suspendieren. Ein Thread kann dabei in der Suspensionsliste mehrerer Variablen vorkommen. Beim Binden einer Variablen werden alle Threads ihrer Suspensionsliste geweckt.
- Jeder Ausdruck der Quellsprache wird in eine Sequenz von Maschineninstruktionen übersetzt.
- Die Implementierung von *Case* ist nicht trivial, da die Subsumption oder Dissubsumption des Wächters entschieden werden muß. So muß es möglich sein, lokale von globalen Variablen unterscheiden zu können. Zudem müssen globale Variablen temporär gebunden werden können, so daß die Bindung eventuell wieder zurück genommen werden kann.
- Inkrementelle Übersetzung in einer interaktiven Entwicklungsumgebung und die getrennte Übersetzung von Modulen können leicht durch Einführen einer *Import-* und einer *Exporttabelle* erreicht werden.
- Das Format für Maschinenprogramme ist leistungsfähig genug, so daß es leicht für die Bedürfnisse von *Persistenz* und *verteilter Programmierung* erweitert werden kann.

Kapitel 4

Optimierungen

In Kapitel 3 haben wir eine Maschine für L vorgestellt. Diese Maschine erlaubt zwar eine vollständige, korrekte Implementierung der Sprache, allerdings lag hier der Schwerpunkt darauf, ein möglichst übersichtliches Gesamtbild zu vermitteln. Daher sind die dort vorgestellten Techniken noch vielfach zu naiv, so daß eine direkte Implementierung des Modells aus dem vorangehenden Kapitel unter dem Gesichtspunkt der Effizienz betrachtet sicher nicht kompetitiv mit anderen vergleichbaren Programmiersprachen wäre.

Wir wollen daher in diesem Kapitel eine Reihe von Optimierungstechniken vorstellen. Dabei handelt es sich um Techniken, die sich zum Teil auch in anderen Programmiersprachen gut bewährt haben, und die direkt (unter Umständen mit mehr oder weniger großen Modifikationen) auch auf L angewandt werden können. Wir werden aber auch verschiedene Techniken vorstellen, die in dieser Form nur aufgrund spezieller Eigenschaften von L anwendbar sind (beispielsweise die Technik der dynamischen Codespezialisierung, vgl. Abschnitt 4.1).

Die in diesem Kapitel beschriebenen Techniken bewegen sich auf dem Abstraktionsniveau der Maschine, wie sie im vorangehenden Kapitel definiert wurde. Um aber eine wirklich hochperformante effiziente Implementierung zu erreichen, bedarf es auch noch einiges an Arbeit, die auf tieferen Abstraktionebenen zu leisten ist. Auf diese Techniken werden wir in Teil III dieser Arbeit eingehen.

4.1 Codespezialisierung

Viele der Optimierungen, die wir im folgenden beschreiben werden, lassen sich als Instanz eines allgemeineren Optimierungsprinzips verstehen, der *Codespezialisierung*. Wie wir in Kapitel 3 gesehen haben, muß die Maschine bei der Ausführung vieler Instruktionen in der Regel eine Reihe von Tests durchführen. So müssen beispielsweise manche Argumente zunächst dereferenziert werden, danach wird dann etwa geprüft, ob es sich bei bestimmten Argumenten um noch ungebundene Variablen handelt, um dann zu suspendieren; ist das Argument gebunden, so muß dann üblicherweise noch getestet werden, ob es vom richtigen Typ ist. Dem können sich noch weitere Tests anschließen, wie beispielsweise der Test auf korrekte Stelligkeit bei der Funktionsapplikation.

In vielen dieser Fälle kann allerdings zugesichert werden, daß die Argumente einer bestimmten Instruktion I bei ihrer Ausführung zur Laufzeit gewisse Eigenschaften erfüllen werden. In die-

sen Fällen ersetzt man dann I durch eine speziellere Variante I', die zur Laufzeit diese Tests dann nicht mehr durchführen muß und daher entsprechend effizienter ausgeführt werden kann. Je nach Anzahl der zu testenden Argumente von I und nach den Kriterien der Tests (Dereferenziertheit, Typ, etc.) können sich daher sehr viele spezielle Varianten von I ergeben. Grundsätzlich muß man daher prüfen, ob die Implementierung spezieller Varianten einer Instruktion überhaupt lohnt, das heißt, ob der Geschwindigkeitsvorteil hinreichend groß und ob die Instruktion auch oft genug verwandt wird. Denn auf der anderen Seite ist es wünschenswert, den Instruktionssatz möglichst klein zu halten, um Wartungsarbeit zu sparen und das Cachingverhalten des Emulators zu verbessern.

Man könnte aber auch so vorgehen, daß man die Dereferenzierung und das Typtesten nicht von jeder Instruktion selbst vornehmen läßt, sondern dafür eigene Instruktionen vorsieht. Dies wäre bei nativer Codeerzeugung auch sicher sinnvoll. Bei einem Emulator wird die Einsparung bei den Sonderfällen aber von dem Mehraufwand zur Interpretation dieser Mini-Instruktionen in den anderen Fällen mehr als egalisiert.

Je nachdem, wann man eine Ersetzung durchführt, unterscheiden wir zwischen *statischer* und *dynamischer* Code-Spezialisierung.

4.1.1 Statische Spezialisierung

Wir wollen das Vorgehen bei der statischen Spezialisierung im folgenden anhand eines konkreten Beispiels beschreiben. Wir betrachten das folgende Programm:

```
let val g = lvar();
    con A;
    fun f(x,y) =
      unify(x,A);
      unify(y,g)
in ... end
```

Wir wollen im folgenden den Code genauer betrachten, der für den Rumpf von `f` erzeugt wird. Im unoptimierten Fall würde dieser wie folgt aussehen:

```
unify X0 V22
unify X1 V23
return unit
```

Hier wird für jede Unifikation einfach eine Instruktion `unify` eingesetzt. Für die freien Variablen `A` und `g` haben wir im Beispiel angenommen, daß sie über die virtuellen Register 22 und respektive 23 zugegriffen werden, sprich daß der Code an dieser Stelle einen direkten Verweis auf den entsprechenden Knoten enthält. Die Instruktion `unify` ist in der Praxis recht teuer: beide Argumente müssen dereferenziert werden, für jedes Argument muß festgestellt werden, ob es sich um eine Variable handelt, ist keines der Argumente eine Variable, so müssen die Typen beider Knoten besorgt und verglichen werden und davon abhängig weitere Aktionen ergriffen werden. Bei der ersten Unifikation `unify(x,A)` ist dieser Aufwand allerdings nicht notwendig, da bereits statisch bekannt ist, daß das zweite Argument ein Konstruktor ist. Man ersetzt daher die erste Instruktion statisch zur Übersetzungszeit im Rumpf von `f` durch eine neue spezialisierte Instruktion

```
unifyConst X0 V22
```

Diese kann nun davon ausgehen, daß das zweite Argument direkt (also ohne Umweg über ein Register) auf einen Knoten verweist, für den Gleichheit über die Gleichheit von Knoten definiert ist (also Konstruktoren, Funktionen und Zellen). Die Instruktion `unifyConst` arbeitet daher wie folgt: das erste Argument wird dereferenziert. Dann wird geprüft, ob der resultierende Knoten derselbe ist, auf den das zweite Argument verweist. Wenn nicht, wird dann noch getestet, ob es sich um einen Variableknoten handelt, der dann gebunden wird. So kann eine vergleichsweise teuere Unifikation durch eine sehr viel einfachere Instruktion ersetzt werden.

4.1.2 Dynamische Spezialisierung

Die statische Spezialisierung kommt vorwiegend dann zum Einsatz, wenn im Quellprogramm *Konstanten* wie Konstruktoren oder Zahlen vorkamen, und daher gewisse Eigenschaften zugesichert werden können. Dagegen dient die dynamische Codespezialisierung dazu Optimierungen aufgrund der Werte von *globalen Variablen* vorzunehmen, die statisch nicht bekannt sind. Hier wird die Ersetzung von Instruktionen nicht bereits zur Übersetzungszeit vorgenommen, sondern erst später *zur Laufzeit*, dann also, wenn größtmögliche Information über alle Parameter ultimativ verfügbar ist.

Wir wollen das Vorgehen an Hand der zweiten Unifikation `unify(y,g)` aus obigem Beispiel betrachten, die ebenfalls zunächst zu einer `unify` Instruktion übersetzt wird. Hier ist a priori nicht klar, ob `g` zur Laufzeit wie im ersten Beispiel an einen Konstruktor gebunden sein wird. Trotzdem besteht hier Spielraum zur Optimierung, was an einer wichtigen zentralen Eigenschaft der Sprache liegt, der *Monotonie-Eigenschaft*: da es sich bei `g` um eine freie Variable von `f` handelt und `g` zudem auf Toplevel (das heißt nicht im Rumpf einer anderen Funktion) definiert wurde, ist per Konstruktion der Sprache L folgendes sichergestellt: wenn `g` auf einen Nichtvariable-Knoten verweist (modulo Dereferenzierung versteht sich), dann kann sich der Knoten auf den `g` zeigt (oder gar dessen Markierung) nie mehr (im Gegensatz zu imperativen Sprachen) ändern. Lediglich, wenn `g` einen `VAR`-Knoten referiert, kann sich dieser im Laufe der Berechnung noch verändern, allerdings auch nur solange, bis eine Bindung an eine Nichtvariable erfolgt ist.

Diese wichtige Eigenschaft können wir uns zu Nutze machen, indem wir die Instruktion

```
unify X1 V23
```

aus obigem Beispiel durch eine neue Instruktion

```
specUnify X1 V23
```

ersetzen. Diese neue Instruktion arbeitet nun so, daß sie nach Prüfung ihrer Argumente sich selbst durch eine spezialisierte Variante *ersetzen* wird. Es handelt sich hierbei also um selbstmodifizierenden Code. Die Spezialisierung erfolgt grundsätzlich nur in Abhängigkeit von Argumenten, die V-Register verwenden, da sich die Werte von X-, L- und auch G-Argumenten ja bei jeder Ausführung der Instruktion verändern können.

Je nach dem Wert von V_{23} wird `specUnify` sich selbst durch unterschiedliche Instruktionen ersetzen. In obigem Beispiel könnte die Instruktion `specUnify` sich selbst zu

```
unifyConst X1 deref(V23)
```

ersetzen, falls $\texttt{V}_{23}$ beispielsweise auf einen CON-Knoten verweist, was in der Praxis, so wie Konstruktoren eingesetzt werden, sehr häufig vorkommt. Wichtig ist, daß bei dieser Ersetzung auch sofort eventuell vorhandene Referenzketten im zweiten Argument aufgelöst werden (im Beispiel durch einen Aufruf der Funktion `deref` angedeutet), da `unifyConst` im zweiten Argument keine Verweise auf `REF`-Knoten erlaubt. Wenn $\texttt{V}_{23}$ auf ein Tupel verweist, könnte man eine Instruktion `unifyTuple` implementieren; allerdings werden die Vorteile der Einführung einer solchen Instruktion in der Praxis nicht sehr hoch sein. Hier ist es sicher einfacher, sich die Einführung einer zusätzlichen Instruktion zu sparen und stattdessen `specUnify` durch das allgemeine `unify` zu ersetzen. Enthält $\texttt{V}_{23}$ schließlich eine noch ungebundene Variable, so kann `specUnify` einfach eine normale Unifikation mit $\texttt{X}_1$ durchführen und mit der Ersetzung noch bis zum nächsten Aufruf warten.

4.1.3 Zeitaufwand

Man beachte, daß eine Instruktion, die sich selbst ersetzen wird, nicht zeitkritisch ist, da sie in der Regel höchstens ein einziges Mal ausgeführt wird. So kann sich also eine solche Instruktion ruhig etwas Zeit bei der genauen Prüfung ihrer Argumente lassen und auch der Ersetzungsvorgang an sich muß nicht unbedingt sehr effizient realisiert werden. Liegt die Instruktion innerhalb einer häufig aufgerufenen Funktion, so wird sich dieser Aufwand durch die resultierende effizientere Instruktion schnell amortisieren. Aber auch wenn die neue Instruktion nur selten oder gar nicht ausgeführt wird, zeigt die Praxis, daß der Aufwand für das Ersetzen sich in der Gesamtlaufzeit eines Programmes nicht bemerkbar macht.

4.1.4 Unterschiedliche Instruktionslängen

Wir wollen die konkrete Darstellung des Codebereichs im Speicher zwar auf des Implementierungskapitel aufschieben, wollen an dieser Stelle aber auf ein Problem im Zusammenhang mit der dynamischen Codespezialisierung eingehen. Der Codebereich wird als Feld dargestellt, bei dem Sprungadressen als relative oder absolute Adressen innerhalb dieses Feldes dargestellt sind. Bei der Darstellung der verschiedenen Instruktionen, ist es in der Regel nicht sinnvoll und auch gar nicht möglich, daß alle Instruktionen die gleiche Länge haben. Vielmehr hängt die Länge einer Instruktionen von der Anzahl der Argumente und eventuell auch noch vom Typ der einzelnen Argumente ab. Will man nun eine Instruktion durch eine andere ersetzen, die länger oder kürzer ist, so ändern sich dadurch die Sprungmarken des nachfolgenden Codes. Da Sprünge nur innerhalb von Segmenten erfolgen können, müssen daher mindestens alle Instruktionen des zugehörigen Segments, die diese Marken verwenden, angepaßt werden. Dies ist allerdings sehr aufwendig.

Ein anderes Vorgehen ist hier sinnvoller: wenn die zu ersetzende Instruktion I kürzer ist, als die neue Instruktion I', dann kann man I durch die Hinzunahme von Dummy-Argumenten künstlich verlängern. Ist I dagegen länger als I', dann bestehen grundsätzlich zwei verschiedene Möglichkeiten: zum einen kann man wie zuvor I' künstlich verlängern, was allerdings den Nachteil hat, daß die Länge des Codes dadurch unnötig zunimmt. Alternativ dazu kann man auch mehrere Argumente $A_1, \ldots, A_n$ in einem Argument A zusammenfassen, indem man die A_i auslagert, das heißt man speichert $A_1, \ldots, A_n$ in einer Tabelle und läßt A darauf verweisen. Nach Ausführung und Ersetzen von I kann der entsprechende Tabelleneintrag dann wieder freigegeben werden.

4.1.5 Ersetzung von Instruktionsfolgen

Man kann die oben beschriebene Technik der dynamischen Codespezialisierung auch noch weiter treiben, indem man nicht nur einzelne Instruktionen durch spezielle Varianten ersetzt, sondern ganze Sequenzen von Instruktionen durch neue Sequenzen substituiert, indem man zum Beispiel Funktionen erst zur Laufzeit inlined [HU94] oder noch aggressivere Optimierungen anwendet [Fra97, Kis97]. Um hier zu guten Ergebnissen zu kommen, muß man dann aber mindestens auch eine Neuberechnung der Registerallokation und eventuell auch anderer Übersetzungsvorgänge vornehmen. Somit muß fast das gesamte Backend des Compilers einschließlich der Registerallokation auch noch zur Laufzeit zur Verfügung stehen und die Maschine muß zudem in der Lage sein, diesen anzusprechen.

Dann kann man aber auch gleich dazu übergehen, die gesamte Compilation einer bestimmten Funktion f bis zur Laufzeit aufzusparen. Man speichert den Rumpf von f in einem geeigneten Format, so daß man auf das Frontend des Compilers verzichten kann. Den Rumpf übersetzt man zur Laufzeit in Kenntnis der Werte aller globalen Variablen von f, sobald die Definition von f ausgeführt wird. Dies ist allerdings in der Regel nur dann sinnvoll, wenn f auf Toplevel definiert wird, also nicht innerhalb einer anderen Funktion, da nur dann die Definition von f höchstens einmal ausgeführt wird. Andernfalls würde ja dann auch jedesmal der Code für den Rumpf von f neu erzeugt, was nur in extremen Ausnahmefällen sinnvoll sein kann.

Aufgrund der oben genannten Schwierigkeiten wurden diese Techniken bisher noch nicht in Mozart integriert.

4.1.6 Kopieren von Code

Wir wollen nun eine letzte Möglichkeit der Anwendung der Codespezialisierung näher betrachten. Bei dieser Variante wird für bestimmte ausgewählte Funktionen jedesmal, wenn eine Instanz dieser Funktion erzeugt wird, auch immer eine *Kopie des Codes* der Funktion erzeugt. Dieses Vorgehen wirkt auf den den ersten Blick nicht sehr sinnvoll und gewinnbringend, da ja gerade eine der wichtigen Ideen bei der Implementierung von Funktionen darin liegt, daß unterschiedliche Instanzen ihren Code teilen. Dennoch wird gerade diese Technik in der Praxis in Mozart häufig angewandt: sie kommt in Zusammenhang mit Modulen und der getrennten Übersetzung zum Einsatz.

Wir wollen das Vorgehen an einem Beispiel verdeutlichen. Nehmen wir an, wir wollen ein Modul `MyModule` zur Verfügung stellen, das die Funktionen `foo` und `bar` exportiert. mit den bescheideneren Möglichkeiten von L repräsentieren wir das Modul einfach als Tupel, das als Argumente die beiden Funktionen enthält. Eine mögliche Implementierung könnte dann so aussehen:

```
let fun foo(x)   = ... goodStuff(...) ...;
    fun bar(x,y) = ... openFile(...) ...
in
    save(MyModule(foo,bar),file)
end
```

Hier wird am Ende das Modul mittels der System-Funktion `save` in einer Datei gesichert (vergleiche hierzu auch Abschnitt 3.11), so daß das Modul von dort von Benutzern geladen werden kann, ohne daß diese das Modul jeweils neu übersetzen müssen. Beim Sichern mittels `save`

wird nun nicht nur der Maschinencode von `foo` und `bar` abgespeichert, sondern die kompletten Funktionen, also insbesondere auch die Werte der freien Variablen (und dann so weiter, also der vollständige transitive Abschluß): das sind einmal `goodStuff`, von dem wir annehmen wollen, daß es aus einem anderen benutzerdefinierten Modul stammt und zum anderen `openFile` aus dem Ein-/Ausgabemodul.

Hier liegen nun auch zwei Schwachpunkte dieses Vorgehens: zum einen wird lediglich eine Kopie von `goodStuff` gesichert. Wenn nun die Implementierung von `goodStuff` verbessert wird, wird diese Änderung in unserem Modul nicht sichtbar, es müßte erst neu übersetzt und wieder gesichert werden, womit die Vorteile getrennter Übersetzung verloren gehen. Ein weiterer Nachteil liegt darin, daß auch eine Kopie der System-Funktion `openFile` gesichert wird: wenn dies technisch überhaupt möglich ist, dann ist dies dennoch praktisch wertlos, da diese Funktion plattformspezifisch ist; ein Benutzer des Moduls verlangt aber, daß seine eigene Version von `openFile` verwandt wird und nicht die für ihn in der Regel unbrauchbare Version desjenigen, der das Modul gesichert hat.

Zur Lösung dieser Probleme bietet L bereits genug Abstraktionsmöglichkeiten. Anstatt direkt das Modul zu sichern, sichert man eine Funktion, die dieses Modul erst erzeugen wird und die über die externen Referenzen des Moduls abstrahiert:

```
fun makeMyModule(goodStuff,openFile) =
  let fun foo(x)   =  ... goodStuff(...) ...;
      fun bar(x,y) = ... openFile(...) ...
  in
      MyModule(foo,bar)
  end;
save(makeMyModule,file)
```

Gesichert wird nun also die zweistellige Funktion `makeMyModule`. Wer das Modul verwenden will, muß nun nach dem Laden zuerst diese Funktion noch aufrufen. Dabei muß er als Eingabeparameter die darin importierten Funktionen `goodStuff` und `openFile` zur Verfügung stellen und erhält als Ausgabe das eigentliche Modul.

Die Funktion `makeMyModule` wird auch als *Funktor* bezeichnet, in Anlehnung an die Funktoren aus SML [MTHM97]. Da wir uns hier auf einen Implementierungsaspekt von Funktoren konzentrieren wollen, haben wir uns auf eine deutlich vereinfachte Darstellung von Funktoren beschränkt. Mozart bietet neben syntaktischer Unterstützung noch eine Reihe weiterer Hilfen zum Arbeiten mit Funktoren, die das Realisieren gerade von größeren Projekten deutlich erleichtern, auf die wir aber an dieser Stelle nicht weiter eingehen werden.

Durch die Einführung von Funktoren ist nun zwar die Aufteilung einer Applikation in Module und deren getrennte Übersetzung möglich geworden, allerdings gingen dadurch auch einige wichtige Möglichkeiten für Optimierungen verloren; in der Praxis zeigte sich nach der Umstellung auf Funktoren eine Laufzeiteinbuße von 30 Prozent im Mittel! Grund hierfür ist die Tatsache, daß die Codespezialisierung für Funktoren in vielen Fällen nicht mehr anwendbar ist, da insbesondere die dynamische Spezialisierung nur in Zusammenhang mit Variablen durchgeführt werden kann, die auf Toplevel deklariert wurden. Wir werden beispielsweise in Abschnitt 4.7 sehen, daß die Applikation einer Funktion dann deutlich beschleunigt werden kann, wenn die Variable auf Toplevel definiert wurde. Im Beispiel oben wäre das etwa die Applikation von `goodStuff` innerhalb von `foo`. Da nun aber `goodStuff` als Eingabeparameter für `makeMyModule` dient, wäre es nun falsch, den Code für `foo` in Abhängigkeit vom aktuellen Wert von `goodStuff` zu

verändern: ein weiterer Aufruf von `makeMyModule` mit einem nun verschiedenen Wert für `goodStuff` würde für eine der beiden Instanzen von `foo`, die ja beide den gleichen Code teilen, zu einem falschen Resultat führen.

Die Lösung aus diesem Dilemma heißt *Kopieren von Code*. Wenn man in unserem Beispiel jedesmal bei der Erzeugung einer neuen Instanz von `foo` auch gleich eine Kopie des Codes von `foo` erzeugt, dann ist es auch korrekt, die Codespezialisierung nicht nur anhand von V-Registern sondern auch auf Basis der aktuellen Werte der *G-Register* also aller globalen Variablen einer Funktion durchzuführen. Wir führen daher eine Instruktion `funCopy` als Variante der Instruktion `fun` aus Abschnitt 3.8.3 ein:

`funCopy` $R\ n\ k\ S$
: Erzeugt analog der Instruktion `fun` einen neuen Funktions-Knoten mit Arität n und k freien Variablen und legt eine Referenz darauf in R. Anders als `fun` wird aber auch zusätzlich eine *Kopie* des Codesegmentes S angefertigt und im neu erzeugten Funktions-Knoten gespeichert.

Der Compiler ist nun frei in der Wahl, welche Variante er für eine bestimmte Funktion verwendet. In der Regel wird er die kopierende Version für alle Funktionen eines Funktors einsetzen, in deren Rumpf eine Spezialisierung anwendbar ist, was für sehr viele Funktionen zutrifft. Dies gilt insbesondere nicht nur für die Toplevel-Funktionen, sondern auch für tiefer geschachtelte Funktionen. Dies ist allerdings problematisch, wie folgendes Beispiel zeigt, wenn es als Rumpf eines Funktors verwendet wird:

```
fun f(x,y) =
  let fun g(u,v) = ... goodStuff(...) ...
  in ...
  end
```

So soll auch die Applikation von `goodStuff` innerhalb von `g` über Codespezialisierung optimiert werden. Es ist nun allerdings nicht sinnvoll neben `f` auch für `g` die `funCopy`-Instruktion zu verwenden, da dann ja bei *jedem* Aufruf von `f` eine Kopie des Codes von `g` erzeugt würde. Dieses Problem kann man umgehen, indem man eine zusätzliche Instruktion einführt: `funCopyOnce` arbeitet zunächst genau wie `funCopy`, ersetzt sich nach einmaligem Aufruf dann allerdings durch `fun`. Alternativ dazu kann man auch `funCopy` selbst so modifizieren, daß beim Kopieren auch stets Kopien von allen erreichbaren Codesegmenten angefertigt werden. So können Spezialisierungen vorgenommen werden, ohne daß zusätzliche Kopien angefertigt werden müssen.

Im allgemeinen bedeutet das Kopieren von Code eine deutliche Erhöhung des Speicherverbrauchs. Im Zusammenhang mit Funktoren trifft dies in der Praxis allerdings nicht zu. Funktoren werden nämlich in der Regel nur ein einziges Mal appliziert: denn auch wenn ein bestimmtes Modul innerhalb einer größeren Applikation mehrfach in verschiedenen anderen Modulen importiert wird, so muß dennoch nur eine einzige Instanz dieses Moduls beim Zusammenbau der Applikation erzeugt werden. Somit ergibt sich schlimmstenfalls eine Verdopplung der Codegröße einer Applikation. Allerdings wird in der Regel das Original nach der Instantiierung nicht mehr benötigt, das heißt alle Referenzen darauf gehen verloren. Aus diesem Grund wurde in Mozart die Speicherbereinigung auch auf den Codebereich erweitert, so daß sich in der Praxis keine merkliche Zunahme der Größe des Codebereichs durch das Kopieren von Code einstellt.

4.2 Unifikation

Im Zusammenhang mit Spezialisierung haben wir bereits in Abschnitt 4.1 Optimierungen für Unifikationen vorgestellt. Hier kann eine Unifikation statisch oder dynamisch mittels `specUnify` zu `unifyConst` übersetzt werden.

`unifyConst` R V
: Die Instruktion dereferenziert R. Falls R eine Variable enthält, wird diese an den Inhalt des V-Registers V gebunden. Andernfalls wird getestet, ob R und V auf den selben Knoten im Speicher verweisen.

Eine leichte Abwandlung von `unifyConst`, stellt die Instruktion `unifyInteger` dar, die auf Zahlen arbeitet. Grund hierfür ist, daß gleiche Zahlen nicht notwendig durch den selben Knoten im Speicher dargestellt werden:

`unifyInteger` R V
: Die Instruktion erwartet im V-Register V eine Referenz auf einen Zahlknoten. Falls R nach Dereferenzierung eine Variable enthält, wird diese an den Inhalt von V gebunden. Andernfalls wird getestet, ob R auf einen Zahlknoten verweist, der den gleichen Wert wie V hat.

4.3 Tupelkonstruktion

Bei einer Applikation der Form $x(e_1, \ldots, e_n)$ kann meist bereits statisch sichergestellt werden, daß es sich bei x um einen Konstruktor handelt. In diesem Fall muß dann nicht, wie in Abschnitt 3.8.3 beschrieben, die Instruktion `apply` eingesetzt werden, vielmehr kann das Tupel dann mittels einer neuen Instruktion `newTuple` gefolgt von `setArg` erzeugt werden:

`newTuple` R R' n
: Die Instruktion erwartet in R' einen Konstruktor-Knoten und erzeugt ein neues Tupel mit Marke R' und n uninitialisierten Argumenten und legt einen Verweis darauf in R ab.

Wir wollen das Vorgehen am Beispiel verdeutlichen: eine geschachtelte Tupelkonstruktion der Form

```
let val x = F(G(23),y) in ... end
```

wird zunächst durch Einführung einer Hilfsvariable `z` verflacht:

```
let val z = G(23);
    val x = F(z,y)
in ... end
```

und kann dann zu (`F` und `G` seien als Konstruktoren bekannt)

```
newTuple R_z G 1
setArg R_z 0 23
newTuple R_x F 2
setArg R_x 0 R_z
setArg R_x 1 R_y
```

übersetzt werden.

4.3.1 Argumentzeiger

Der Aufbau eines neuen Tupels der Form

```
let val x=F(u,v) in ... end
```

geschieht wie im vorangehenden Kapitel beschrieben unter Verwendung der Instruktionen `newTuple` und `setArg`:

```
newTuple R_x F 2
setArg R_x 0 R_u
setArg R_x 1 R_v
```

Warren hat hier gleich eine Optimierung integriert, die es erlaubt, zwei Argumente von `setArg` einzusparen, indem er ein Register den sogenannten *Argumentzeiger* (structure pointer) `ap` einführt. `newTuple` wird dabei so angepaßt, daß nach der Erzeugung des Tupel-Skeletts `ap` auf das erste (noch uninitialisierte) Argument zeigt. `setArg` erfährt dann über `ap` wohin sein Argument zu schreiben ist:

`setArg` R
: Schreibt den Inhalt von R in das Argument eines Tupels, auf den der Argumentzeiger `ap` verweist und inkrementiert `ap`, so daß `ap` auf das folgende Argument verweist.

Der Code von oben, würde damit dann wie folgt aussehen:

```
newTuple R_x F 2
setArg R_u
setArg R_v
```

Analog zu `setArg` kann die Instruktion `moveArg`, die zur Unifikation von Tupelargumenten in Wächtern verwandt wird, vereinfacht werden, indem auch `getTuple` `ap` richtig setzt:

`moveArg` R
: Die Instruktion kopiert den Inhalt des Argumentes, auf das `ap` verweist, ins Register R und inkrementiert ap im Anschluß.

Durch die Einführung des Argumentzeigers verringert sich nicht nur die Codegröße. Gerade bei der Verwendung eines Emulators reduziert sich auch der Dekodieraufwand der Instruktion, da ja jedes Argument der Instruktion erst aus dem Speicher geladen werden muß. Zudem kann im Beispiel das Register $R_{\mathtt{x}}$ früher freigegeben und wiederverwandt werden.

4.3.2 Modus-Register

Im vorangegangenen Kapitel haben wir erklärt, daß ein Wächter der Form

```
case x of
  F(y,z) => ...
```

zu folgender Codesequenz übersetzt wird:

```
  getTuple Rx F 2 L1
  moveArg Ry
  moveArg Rz
  branch L2
L1:
  newVar Ry
  setArg Ry
  newVar Rz
  setArg Rz
L2:
  ...
```

Je nachdem ob `x` bereits gebunden ist oder nicht wird mittels `getTuple` entweder zu einer lesenden oder zu einer schreibenden Codefolge gesprungen. Die WAM faßt diese beiden Codeströme zu einem zusammen, indem sie ein neues Register das *Modus-Register* einführt. Dieses kann entweder den Wert `read` oder `write` annehmen und wird von `getTuple` entsprechend gesetzt. Der schreibende Codestrom wird gestrichen (damit kann auch das letzte Argument von `getTuple` wegfallen), seine Funktion wird von `moveArg` übernommen:

```
getTuple Rx f 2
moveArg Ry
moveArg Rz
```

Die Instruktion `moveArg` ist dann wie folgt definiert (hier unter Verwendung des Argumentzeigers):

`moveArg` R
: Wenn `mode` den Wert `write` hat, erzeugt die Instruktion einen neuen Variableknoten und legt eine Referenz darauf sowohl in R ab als auch in dem Argument, auf das `ap` zeigt.
Wenn `mode` den Wert `read` hat, wird das Argument, auf das `ap` zeigt, nach R geladen. Anschließend wird ap inkrementiert.

Auch die Einführung des Modus-Registers verkürzt die Codegröße merklich. Allerdings muß nun von `moveArg` jeweils ein zusätzlicher Test von `mode` durchgeführt werden. Dieser Test fällt in der Regel bei einem Emulator nicht weiter ins Gewicht. Hochperformante native Implementierungen greifen aus diesem Grund allerdings lieber auf die Zweistrom-Technik zurück [VR90, TCS95].

4.3.3 Lese/Schreib-Unifikation außerhalb von Wächtern

Wird eine Unifikation der Form

```
unify(x,F(u,v))
```

außerhalb eines Wächters verwandt, so haben wir dies im vorangegangen Kapitel naiv behandelt, indem man zunächst ein neues Tupel aufbaut und dieses dann mit `x` unifiziert. Dies ist aber dann suboptimal, wenn `x` bereits an ein Tupel gebunden ist. Man kann daher die Technik der Lese/Schreib-Unifikation, auch auf solche Unifikationen anwenden. So kann man dann auch obige Unifikation unter Verwendung von `getTuple` übersetzen. Dadurch wird nur dann neuer Speicher alloziert, wenn dies wirklich unumgänglich ist.

4.3.4 Listen

Listen (also zweistellige Tupel mit der Marke `::`) stellen die mit deutlichem Abstand am häufigsten verwandte Art von Tupeln dar (siehe auch Tabelle 10.12). Daher verwenden wir eine optimierte Darstellung für Listen. Wir führen dazu einen neuen Knotentyp mit Marke `LIST` ein.

Abbildung 4.1 Konventionelle und optimierte Darstellung der Liste `[23]`.

Abbildung 4.1 zeigt vergleichend die optimierte Darstellung einer Liste bestehend aus nur einem Element `23` und der konventionellen Darstellung.

Aus der neuen Darstellung ergeben sich zwei Vorteile: Listen benötigen weniger Speicherplatz als Tupel, da weder die Marke noch die Arität explizit gespeichert werden müssen; im Implementierungsteil in Abschnitt 7.3.4 werden wir sehen, daß sich damit die Größe einer Listenzelle von vier auf zwei Worte reduziert. Darüberhinaus wird auch der Vergleich von Listen einfacher und damit schneller: anstatt nach dem Test auf `TUPLE` noch einzeln Arität und Marke zu vergleichen, reicht hier lediglich der Test, ob beide Knoten mit `LIST` markiert sind.

Die Instruktionen, die Tupel erzeugen können, müssen nun allerdings angepaßt werden: so muß `apply` nun prüfen, ob ein zweistelliges Tupel mit Marke `::` zu erzeugen ist, und dann keinen `TUPLE` sondern einen `LIST` Knoten erzeugen. Die spezialisierte Variante `newTuple` muß diesen Test allerdings nicht durchführen: man kann nämlich jedes Vorkommen von `newTuple` R `'::' 2` durch eine neue Instruktion `newList` R ersetzen:

`newList` R
: Erzeugt einen neuen `LIST` Knoten und legt eine Referenz darauf nach R. Zudem wird `ap` auf das erste Argument der Liste gesetzt.

Analog wird mit `getTuple` verfahren; auch hier wird eine neue Instruktion `getList` definiert:

Tabelle 4.1 Anzahl statisch optimierbarer Erzeugungen von Namen und Anzahl dynamisch erzeugter Namen in Mozart.

Anwendung	Aufrufe statisch		Aufrufe
	total	optimiert	dynamisch
Explorer	70	70	0
Browser	281	281	0
Gump	33	33	0
Compiler	253	140	353
Scheduler	23	23	0
Spedition	13	13	0

`getList` R

Wenn R auf eine Variable verweist, wird ein neuer `LIST` Knoten erzeugt und R daran gebunden. `mode` wird auf `write` gesetzt.
Wenn R auf eine Liste zeigt, wird `mode` auf `read` gesetzt.
In beiden Fällen wird `ap` stets auf das erste Argument der Liste gesetzt.

4.4 Konstruktoren

Bei Konstruktoren kann man grundsätzlich zwei Arten der Verwendung unterscheiden. Zum einen können Konstruktoren dynamisch zur Laufzeit eines Programmes innerhalb einer Funktion mehrfach erzeugt werden. Zum anderen werden Konstruktoren auf Toplevel statisch erzeugt: hier werden Konstruktoren (wie bereits in Abschnitt 2.13.2 beschrieben) in der Regel dazu verwandt, die Sichtbarkeit von Informationen dediziert zu steuern, indem man den Skopus der zugehörigen Variable entsprechend einschränkt. In dieser Form ähnelt die Verwendung von Konstruktoren stark der von Atomen (= Zeichenketten-Konstanten) in Oz, wobei die effiziente Implementierung der Verwendung von Atomen wohlverstanden ist. Bei Konstruktoren und Atomen handelt es sich in beiden Fällen eigentlich um Konstanten mit dem einzigen Unterschied, daß die Konstruktoren lediglich über Variablen zugreifbar sind. Demzufolge ist es wünschenswert, daß die Verwendung von Konstruktoren anstelle von Atomen innerhalb eines Programmes keine nachteiligen Auswirkungen auf die Performanz hat. Gerade das kann aber unter Verwendung der dynamischen Spezialisierung erreicht werden: überall dort, wo bei der Verwendung von Atomen besserer Code erzeugt werden kann, ist dies auch dann möglich, wenn stattdessen Toplevel-Konstruktoren eingesetzt werden, indem man dies über dynamische Spezialisierung erreicht. Dies gilt übrigens nicht nur für die eingeschränkte Sprache L sondern auch für volles Oz: ob als Marke von Tupeln und Records, als Features von Records, bei privaten Methoden, etc. immer können Konstruktoren und Atome unter dem Gesichtspunkt der Effizienz gleichwertig verwandt werden.

In vielen Fällen können die Optimierungen von Konstruktoren auch bereits durch den Compiler vorgenommen werden. So muß in der Regel oft gar nicht auf die dynamische Form der Spezialisierung zurückgegriffen werden.

Die Praxis zeigt nun, daß Konstruktoren (= Namen in Oz) fast ausschließlich so verwandt werden, daß die oben diskutierten Optimierungen angewandt werden können, da Konstruktoren nur in den seltensten Fällen innerhalb von Funktionen erzeugt werden. Tabelle 4.1 zeigt die Anzahl der optimierbaren Verwendungen von Namenserzeugungen in Mozart anhand verschiedener An-

wendungen.[1] Die Spalte „total“ gibt die Anzahl der statischen Vorkommen von Namenserzeugungen im Quelltext an. Die nächste Spalte zeigt, wie viele davon optimiert werden können, weil sie auf Toplevel vorkommen. Es zeigt sich, daß dies in 96% bis 100% aller Vorkommen der Fall ist. Darüber hinaus gibt die Tabelle auch Aufschluß über die Anzahl der dynamisch erzeugten Namen, also solcher Erzeugungen, die nicht optimiert behandelt werden. Hier zeigt sich, daß Namen erwartungsgemäß zur Laufzeit nur extrem sparsam erzeugt werden, wenn man dies etwa mit der Anzahl dynamisch erzeugter Variablen (vgl. Tabelle 10.12 auf Seite 205) vergleicht.

4.5 Arithmetik

Auch im Zusammenhang mit Operatoren lassen sich spezialisierte Instruktionen erzeugen, wenn beispielsweise der Wert eines Argumentes bekannt ist. So könnte man folgende Addition

```
let val x = y+123 in ... end
```

unter Verwendung einer spezialisierten Instruktion `plusConst` übersetzen:

`plusConst` R n R'
: Die Instruktion wartet, bis R auf einen Zahl-Knoten verweist, erzeugt dann einen neuen Zahl-Knoten k im Speicher, der die Summe des Wertes von R und der Zahl n enthält und legt anschließend eine Referenz auf k in R' ab.

Die Einsparung betrifft hier den Wegfall des Dereferenzierens, des Typtestens und der Beschaffung des Wertes des zweiten Arguments.

Weitere schwächere Formen der Optimierung sind denkbar, wenn beispielsweise bekannt ist, daß ein Argument vom richtigen Typ ist, aber vielleicht nicht sicher ist, ob es auch bereits dereferenziert ist. Auch der symmetrische Fall (dereferenziert, aber Typ unbekannt) könnte speziell behandelt werden. So kommt man allerdings sehr schnell zu einem runden Dutzend spezieller Versionen für jeden einzelnen Operator, bei denen der Gewinn im Vergleich zum Aufwand sehr fraglich ist.

4.6 Case

In Abschnitt 3.9 haben wir ein Schema zur effizienten Implementierung von Case vorgestellt. Spezielle Arten von Case kommen nun allerdings in der Praxis besonders häufig vor und sind deshalb einer weiteren Optimierung wert.

4.6.1 Einfache Tests

Sehr häufig tauchen Ausdrücke der Form

[1] Eine Beschreibung der verwendeten Programme befindet sich in Anhang B.

```
case x of
     t => e
   | ...
```

auf, bei denen t eine sehr einfache Struktur hat.

Konstanten

Analog zum Vorgehen bei `unifyConst` und `unifyInteger` (vgl. Abschnitt 4.2) kann dann deutlich optimiert werden, wenn es sich bei t um einen Konstruktor oder eine Zahl (statische Spezialisierung) oder um eine Variable handelt, die an eine Zelle, Funktion oder Namen gebunden ist (dynamische Spezialisierung). Man kann in diesen Fällen besonders effizienten Code erzeugen, indem man den ganzen Wächter durch eine einzige spezielle Instruktion `testConst` bzw. `testInteger` realisiert:

`testConst` R V L
: Die Instruktion erwartet im V-Register V einen (dereferenzierten) Verweis auf einen Konstruktor, eine Zelle oder eine Funktion. R wird dereferenziert. Falls R eine logische Variable enthält, wird auf dieser Instruktion suspendiert. Andernfalls wird geprüft, ob R auf den selben Knoten im Speicher wie V verweist. Falls ja, wird die nächste Instruktion ausgeführt, falls nein wird zur Marke L gesprungen.

`testInteger` R V L
: Diese Instruktion arbeitet analog zu `testConst`. Sie erwartet in V allerdings einen Zahl-Knoten, dessen Inhalt mit R verglichen wird.

Ohne diese Optimierung würde der Wächter eines Case der Form

```
case x of 23 => e1 | y => e2
```

wie folgt übersetzt

```
guardStart L
unifyInteger Rx 23
guardEnd
```

Durch die Einführung dieser speziellen Instruktionen spart man also die Ausführung von `guardStart` und `guardEnd` und damit auch das Setzen und Rücksetzen von `caseStart`. Außerdem wird das Testen der Spur beim Erreichen des Endes des Wächters eingespart. Somit reduziert sich das Case zur Laufzeit auf einen einfachen Test des Wertes von `x`, an dem etwa die Spur überhaupt nicht mehr beteiligt ist. Dies zeigt, daß solch einfache Tests sich in L genauso effizient implementieren lassen, wie in anderen Sprachen, obgleich das Case von L im allgemeinen wesentlich mächtiger und ausdrucksstärker ist.

Tupel

Falls es sich bei dem Wächter t um ein Tupel handelt, kann nicht immer so wie im vorangehenden Abschnitt für Konstanten verfahren werden. So ist nur in bestimmten Fällen (die allerdings in der Praxis fast ausschließlich auftreten) auch eine Sonderbehandlung nach obigem Schema möglich. Dann nämlich wenn sichergestellt ist, daß der Test auf Subsumption/Dissubsumption durch eine einzige Instruktion erfolgen kann. Die ist in folgendem Beispiel etwa nicht der Fall:

```
case x of F(y,y) => ...
```

Hier muß nämlich nicht nur getestet werden, ob `x` auf ein einstelliges Tupel mit Marke `F` verweist. Zudem müssen die beiden Argumente von `x` noch auf Gleichheit getestet werden.

Eine leichte Abwandlung des obigen Beispiels, die sehr häufig anzutreffen ist, kann allerdings optimiert werden:

```
case x of F(y,z) => ...
```

Hier sind `y` und `z` lokale ungebundene Variablen, so daß die Unifikation mit den Argumenten von `x` sicher erfolgreich ist. Im allgemeinen kann ein Wächter der Form $f(x_1, \ldots, x_n)$ dann optimiert werden, wenn die x_i alles lokale paarweise verschiedene Variablen sind. In diesem Fall ist der Wächter genau dann subsumiert, wenn x an ein Tupel mit Marke f und Arität n gebunden ist. Die entsprechende Maschineninstruktion heißt dann `testTuple`. Dabei ist zu beachten, daß die Argumente x_i nach erfolgreichem Test noch durch jeweils eine eigene `moveArg` Instruktion in ihre Register gerettet werden müssen und daher die Instruktion `testTuple` dafür sorgen muß, daß der Argumentzeiger `ap` und das Register `mode` zuvor richtig gesetzt werden:

`testTuple` R f n L
: Falls R nach Dereferenzierung eine Variable enthält, wird suspendiert. Andernfalls wird geprüft, ob R ein Tupel mit Marke f und n Argumenten enthält. Falls nein wird zur Marke L gesprungen. Falls ja, werden `mode` auf `read` und `ap` auf das erste Argument gesetzt, danach wird die nächste Instruktion ausgeführt.

Man kann das Setzen und Testen von `mode` in diesem speziellen Fall umgehen, wenn man `moveArg` durch eine spezielle Instruktion `loadArg` ersetzt, die genauso arbeitet wie `moveArg` im `read`-Modus:

`loadArg` R
: Das Argument, auf das `ap` zeigt, wird nach R geladen.

Wenn man die Listenoptimierung aus Abschnitt 4.3.4 mit einbezieht, muß man auch hier für Listen eine eigene Instruktion vorsehen:

`testList` R L
: Falls R nach Dereferenzierung eine Variable enthält, wird suspendiert. Andernfalls wird geprüft, ob R eine Liste enthält. Falls ja, wird nach Setzen von `ap` und `mode` die nächste Instruktion ausgeführt, falls nein wird zur Marke L gesprungen.

Als Beispiel betrachten wird den Maschinencode zu folgendem Ausdruck:

```
case x of
  h::r => f(h); g(r)
| nil  => unit
```

Der entsprechende Maschinencode sieht dann wie folgt aus:

```
  testList R_x L1
  loadArg R_h
  loadArg R_r
  move R_h X0
  apply R_f 1
  move R_r X0
  apply R_g 1
  branch L2
L1:
  unifyConst R_x nil
  move unit X0
L2:
```

4.6.2 Boolesche Tests

Besonders häufig treten Ausdrücke auf, die das Ergebnis des Aufrufs einer booleschen Funktion testen:

```
case f(x,y) of
  true  => ...
| false => ...
```

Hier bietet sich die Einführung einer Instruktion `testBool` analog zu `testConst` an, die testet, ob ihr Argument den Wert **`true`** oder **`false`** hat und sonst eine Fehlermeldung ausgibt. Dadurch wird deutlich Code eingespart (für die Fehlermeldung und durch die Zusammenfassung von zwei Tests in einer Instruktion), was gleichzeitig zu einem Geschwindigkeitsvorteil durch Wegfall des emulatorbedingten Mehraufwandes führt:

`testBool` R L
: Falls R nach Dereferenzierung eine Variable enthält, wird suspendiert. Andernfalls wird geprüft, ob R den Wert **`true`** hat. Falls ja, wird die nächste Instruktion ausgeführt, falls R den Wert **`false`** enthält, wird zur Marke L gesprungen. Andernfalls wird eine Fehlermeldung ausgegeben.

Die Optimierung von booleschen Tests kann für spezielle Fälle noch weiter getrieben werden. Dann nämlich wenn es sich bei der Funktion `f` aus obigem Beispiel um eine spezielle vordefinierte Funktion wie zum Beispiel einen Typtest oder einen arithmetischen Test handelt. So kann man beispielsweise bei einem arithmetischen Test in der Form

```
case x<y of
  true  => ...
| false => ...
```

den Aufruf des Vergleichsoperators mit dem darauffolgenden Test direkt zu einer neuen Instruktion `testLess` zusammenfassen:

`testLess` R R' L
: Falls R oder R' nach Dereferenzierung eine Variable enthalten, wird suspendiert. Andernfalls wird geprüft, ob der Inhalt von R kleiner als der von R' ist. Wenn ja, wird die nächste Instruktion ausgeführt, andernfalls wird zur Marke L gesprungen.

Da < stets **true** oder **false** liefert, spart man auch den Fall für die Fehlerbehandlung. Somit kann man auch diesen in der Praxis sehr häufig auftretenden Fall auf einen simplen Test reduzieren, der in nichts dem Pendant in vergleichbaren Sprachen nachsteht.

4.6.3 Redundante Tests

Oft will man in einem Pattern die Werte verschiedener Variablen gleichzeitig testen, wie das etwa in der folgenden Funktion zur Realisierung eines booleschen Und geschehen ist:

```
fun and(x,y) =
  case [x,y] of
    [true,true] => true
  | _           => false
```

Bei der Übersetzung muß man nun nicht die Liste `[x,y]` vom Code explizit aufbauen lassen, um sie dann später mittels `getList` wieder zu zerlegen. Man kann stattdessen die redundanten Tests auslassen und den Rumpf von `and` gleich wie folgt übersetzen:

```
  guardStart L1
  unifyConst Rx true
  unifyConst Ry true
  guardEnd
  return true
L1:
  return false
```

4.6.4 Indexierung

Eine wichtige Technik zur Optimierung großer Case Ausdrücke der Form

```
case x of
  p1 => e1
| p2 => e2
  ...
| pn => en
```

ist die Technik der *Indexierung*. Die Idee dabei ist, nicht alle Pattern p_i sequentiell nacheinander zu testen, sondern durch einen Tabellenzugriff in Abhängigkeit von x direkt die Menge $\overline{p}$ der Pattern zu bestimmen, die überhaupt in Frage kommen. Oft enthält $\overline{p}$ dabei nur ein Element, andernfalls kann man durch weitere Indizierungsschritte über $\overline{p}$ entsprechend fortfahren.

Zur Übersetzung eines indizierten Case dient die Instruktion `switch`. Dabei verwendet `switch` eine *Hashtabelle*, die als Eingabe einen beliebigen Wert erhält und dazu eine Codeadresse zurückliefert, die dann angesprungen wird.

`switch` R V
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle H. Falls R eine Variable enthält wird suspendiert. Andernfalls liefert H abhängig vom Inhalt von R eine Codeadresse, zu der dann gesprungen wird.

Damit ein V-Register eine Hashtabelle referieren kann, werden die folgenden Preludeinstruktionen definiert:

`vmakeHT` V n
: Erzeugt eine neue Hashtabelle der Größe n und legt eine Referenz darauf im V-Register V ab.

`vaddHTCon` V V' L
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle und speichert in dieser unter dem Konstruktor aus V' die Marke L.

`vaddHTInt` V `n` L
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle und speichert in dieser unter der Zahl n die Marke L.

`vaddHTTuple` V V' n L
: Die Instruktion speichert in der Hashtabelle aus V, daß für ein Tupel mit Marke V' und Arität n zur Marke L zu springen ist.

Ein wirklicher Vorteil durch die Indizierung ergibt sich erst dann, wenn die Tabellen hinreichend groß werden. Die Indizierung eines Case, das nur aus zwei oder drei Pattern besteht, stellt sich als nicht wirkungsvoll heraus. Im Gegenteil: hier sind einige wenige einfache Tests etwa über `testConst` effizienter als ein aufwendigerer Hashing-Schritt. Ab welcher Schachtelungstiefe eine Indizierung Sinn macht, ist in der Praxis schwer zu bestimmen, da die Qualität des Hashens stark von der Art der Eingabe, den in der Tabelle verwandten Schlüsseln und nicht zuletzt von der Wahl der Hashfunktion abhängen. Aber auch bei sehr großer Schachtelungstiefe muß eine Indizierung nicht immer vorteilhaft sein: dann nämlich wenn die Eingabe `x` etwa in 99% aller Fälle immer

denselben Wert hat (was zum Beispiel bei der Iteration über eine Liste der Fall sein kann). Dann ist es sinnvoller zuerst auf diesen Wert zu testen.[2]

Das in Mozart verwandte Indizierungsschema ist noch vergleichsweise einfach gehalten, erweist sich in der Praxis allerdings als durchaus effektiv. Komplexere Verfahren [Bra95, KS90, KS88] behandeln oft Fälle, die entweder selten vorkommen oder aber in Oz durch Umkodierung ebenso effizient implementiert werden können.

Statisch getypte Sprachen sind bei der Indexierung in der Regel etwas im Vorteil: die Konstruktoren eines Datentyps können durch fortlaufende kleine Zahlen dargestellt werden, so daß der Hashing-Schritt durch einen Feldzugriff ersetzt werden kann.

4.7 Funktionen

Für die effiziente Übersetzung von Funktionen wurden in der funktionalen Welt eine Reihe von Techniken entwickelt, die sich (mitunter mit leichten Modifikationen) auf L übertragen lassen. Darüber hinaus, gibt es aber auch verschiedene Optimierungstechniken, die speziell nur im Zusammenhang mit Oz zum Tragen kommen.

4.7.1 Endrekursion

Sprachen, die zwar rekursive Funktionen aber keine Schleifenkonstrukte unterstützen, müssen *Endrekursion* (engl. tail recursion) optimieren. Endrekursion bedeutet, daß der letzte Ausdruck im Rumpf einer Funktion eine Applikation ist. Im unoptimierten Fall würde dabei folgendes Codefragment erzeugt:

```
apply R n
deallocate
return X0
```

Hier wird von `apply` die auf `apply` folgende Instruktion auf dem Keller abgelegt, so daß nach Rücksprung aus der gerufenen Funktion lediglich die aktuelle Umgebung dealloziert und danach mittels `return X0` der nächste Auftrag vom Keller genommen wird. Man kann in diesem Fall die Deallokation der Umgebung vor die Applikation ziehen und die Instruktion `apply` und das nachfolgende `return` durch eine Instruktion `tailApply` ersetzen, die mit `apply` identisch ist, bis darauf daß sie die Folgeinstruktion nicht auf dem Keller ablegt, den Keller also gar nicht verändert.

Dieses Vorgehen ist allerdings nur dann korrekt, falls das Register R nicht die Form $\texttt{L}_i$ hat, da in diesem Falle die zuvor bereits deallozierte Umgebung referiert würde. Dieses Problem läßt sich beheben, indem man $\texttt{L}_i$ vor der Deallokation in ein freies Register kopiert. Da bei der Applikation einer n-stelligen Funktion lediglich die die Argumente enthaltenden Register $\texttt{X}_0$ bis $\texttt{X}_{n-1}$ noch benötigt werden, ist ein solches freies Register zum Beispiel $\texttt{X}_n$:

```
move Li Xn
deallocate
tailApply Xn n
```

[2] Gerade diese Unwägbarkeiten haben uns auch bewogen bei der Implementierung der virtuellen Maschine an performanz-kritischen Stellen oft auf die Verwendung der C++ **`switch`**-Anweisungen zu verzichten und diese stattdessen durch geschachtelte Konditionale zu ersetzen, bei denen die wahrscheinlichsten Fälle zuerst behandelt werden.

Wie oben beschrieben, liegt Endrekursion dann vor, wenn die letzte Anweisung, die innerhalb eines Funktionrumpfes ausgeführt wird, ein Funktionaufruf ist. Das gilt allerdings nicht nur, wenn die letzte Anweisung im Rumpf syntaktisch gesehen eine Applikation ist, wie folgendes Beispiel zeigt:

```
fun f(x) =
  case x of
    1 => g(x)
  | _ => h(x)
```

Auch hier kann in beiden Zweigen des Case Endrekursions-Optimierung vorgenommen werden. Allerdings ist hier Vorsicht geboten, da der unoptimierte Maschinencode zunächst wie folgt aussieht:

```
    ...
    apply R_g 1
    branch L
    apply R_h 1
L:
    deallocate
    return X_0
```

Man muß hier darauf achten, daß man das `deallocate` zuerst dupliziert, indem man es auch in den ersten Zweig hereinzieht und erst dann beide `apply` durch je ein `tailApply` ersetzt.

Man muß beachten, daß durch Endrekursionsoptimierung das Fehlen von Schleifen im allgemeinen nicht völlig ausgeglichen werden kann. So müssen jeweils ein L-Registersatz alloziert und dealloziert werden. Auch müssen die Variablen, die innerhalb der Schleife verwandt werden, entweder explizit als Argumente an die Funktion übergeben werden, die die Schleife realisiert. Das bedingt in der Regel das Einsetzen zusätzlicher `move`-Instruktionen und die Vergrößerung des L-Registersatzes. Alternativ dazu kann man die Variable auch in einen Abschluß einbetten, was aber zusätzlichen Speicher kostet und den Zugriff auf diese Variablen verteuert, da sie nicht mehr in Registern liegen können. Durch eine Wiederverwendung der L-Register der rufenden Funktion durch die gerufene Funktion, kann man versuchen, diese Nachteile auszugleichen [Mei91].

4.7.2 Applikation

Vergleicht man die Implementierung der Applikation in L mit der von statisch getypten Sprachen oder mit Sprachen, die keine höheren Funktionen unterstützen (wie dies zum Beispiel in Prolog der Fall ist), so stellt man fest, daß in L bei jeder Applikation mehr Aufwand nötig ist. So kann in Prolog direkt im Code eine Referenz auf die zu applizierende Funktion abgelegt werden. Dies ist in L im allgemeinen nicht möglich: da Funktionen über Variablen angesprochen werden, müssen hier bei der Applikation von `x` zuerst verschiedene Tests durchgeführt werden: zuerst muß ein Dereferenzierungsschritt durchgeführt werden. Dann muß festgestellt werden, ob es sich bei `x` um eine logische Variable, eine Funktion oder einen Konstruktor handelt. Darauf muß ein Test auf korrekte Arität folgen. Erst dann kann der eigentliche Unterprogrammsprung erfolgen. So gesehen ist L gegenüber vergleichbaren Sprachen in dieser Beziehung zunächst deutlich im Nachteil, vor allem vor dem Hintergrund, daß ein Funktionaufruf wegen fehlender Schleifenkonstrukte einen zentralen sehr häufig genutzten Bestandteil der Sprache bildet.

Wir werden in diesem Abschnitt aber Techniken vorstellen, die zeigen, daß sich dieser Nachteil in sehr vielen Fällen wieder ausgleichen läßt.

Spezialisierung

Auch bei der Funktionsapplikation kann die Technik der Codespezialisierung wichtige Dienste leisten, indem man die oben beschriebenen Tests nur ein einziges Mal statisch durch den Compiler oder dynamisch zur Laufzeit durchführt. Hierzu definieren wir die Instruktionen `fastApply` und `specApply` (die endrekursiven Varianten `fastTailApply` und `specTailApply` sind analog definiert):

`fastApply` V
: Die Instruktion erwartet im V-Register V eine Referenz auf einen Funktion-Knoten f. Sie arbeitet genau wie `apply` ohne allerdings Dereferenzierung, Typtests und Aritätstest durchzuführen.

`specApply` V n
: Die Instruktion suspendiert auf Determiniertheit von V. Verweist V auf einen n-stelligen Funktion-Knoten, so ersetzt sich selbst dann durch `fastApply` V. Verweist V auf einen Konstruktor, dann ersetzt sie sich durch `newTuple` V n.

Mit der Technik der Spezialisierung lassen sich alle Applikationen von auf Toplevel deklarierten Funktionen optimieren. So steht die Implementierung der Applikation in L in diesem Punkt in nichts den Implementierungen in getypten, funktionalen oder auch first-order Sprachen wie Prolog nach. Diese Form der Applikation deckt in der Praxis bereits das Gros aller vorkommenden Applikationen ab. Darüber hinaus lassen sich aber auch einige Fälle von Applikationen von Variablen optimieren, die nicht auf Toplevel definiert wurden, wie folgendes Beispiel zeigt:

```
fun f(x,y) =
  let fun g(u) = e
  in  ...
      g(y)
  end
```

Da hier statisch sicher ist, daß bei der Applikation von `g` sowohl Arität als auch Typ korrekt sind, könnte man die Definition von `fastApply` erweitern, indem man anders als oben definiert neben V-Registern als Argument auch beliebige andere Registerarten zuläßt.

Ein wichtiger Nebeneffekt der Spezialisierung ist auch der, daß die Größe von Abschlüssen deutlich reduziert wird. Da nun der Code direkt auf die Funktionen verweist, müssen diese nicht in die G-Register aufgenommen werden. Da die meisten globalen Variablen einer Funktion f diejenigen Funktionen sind, die f in seinem Rumpf aufruft, kann so der für f zu konstruierende Abschluß deutlich kleiner ausfallen.

Inline-Caching

Somit verbleiben häufig nur noch die echten higher-order Applikation, die nicht so effizient behandelt werden können. Aber auch hier läßt sich noch ein wenig an Laufzeit einsparen, indem man die Technik des *Inline-Caching* [Deu84] auf diesen Fall überträgt.

Wir wollen das Vorgehen am Beispiel erläutern und betrachten die Definition der Funktion `map`, die eine einstellige Funktion `f` auf alle Elemente einer Eingabeliste `l` anwendet und als Ergebnis die Liste der Ausgabewerte zurückliefert:

```
fun map(l,f) =
  case l of
    h::r => f(h)::map(r,f)
  | nil  => nil
```

Hier handelt es sich bei dem Aufruf von `f` im Rumpf von `map` um eine echte higher-order Applikation. Dem Einsatz von Inline-Caching liegt nun folgende Heuristik zu Grunde: obgleich sich prinzipiell bei jedem Aufruf von `map` der Wert von `f` ändern kann, wird er dies in der Praxis nur eher selten tun (was in der Regel von der Länge der Eingabeliste abhängt). Dies kann man sich zu Nutze machen, indem man sich beim Aufruf von `f` den aktuellen Wert statisch merkt. Beim nächsten Mal prüft man nun zuerst, ob `f` noch denselben Wert hat. Wenn ja, kann man sich den Typtest und den Aritätstest sparen. Man kann sich auch das Dereferenzieren sparen, wenn man sich zwei Werte merkt: den Wert von `f` vor und nach Dereferenzierung. Das Merken der beiden Werte kann direkt im Code erfolgen, wenn man eine neue Instruktion `cachedApply` vorsieht, die jeweils dafür ein Argument zur Verfügung stellt, das dann als Cache dient und dynamisch geändert werden kann:

`cachedApply` $R\ n\ V\ V'$
: Die Instruktion prüft, ob R (ohne vorherige Dereferenzierung) und V auf denselben Knoten im Speicher verweisen, In diesem Fall enthält V' einen Verweis auf einen Funktion-Knoten, das heißt die Instruktion arbeitet in diesem Fall genau wie `fastApply` V'.
Wenn R und und V verschieden sind, dann arbeitet die Instruktion zunächst genau wie `apply`. Allerdings wird vor dem Unterprogrammsprung zusätzlich noch das Argument V durch den nicht dereferenzierten Inhalt von R und V' durch den dereferenzierten Inhalt von R ersetzt.

Bei einem Treffer im Cache spart man so zwei Tests ein, die mit zusätzlichen Speicherzugriffen verbunden sind. Im anderen Fall bezahlt man zusätzlich einen Test und zwei schreibende Speicherzugriffe. Somit bringt diese Technik im Zusammenhang mit der Applikation nur eine geringfügige Geschwindigkeitssteigerung mit sich. Die Technik hat sich allerdings in Mozart in anderem Zusammenhang sehr bewährt, weswegen sie auch hier exemplarisch vorgestellt wurde: beim Zugriff auf die Features eines Records [RMS96, Meh99] und bei der Objektapplikation [Hen97] konnten damit signifikante Verbesserungen erzielt werden, weil dadurch aufwendigere Hashingschritte durch einen einfachen Vergleich ersetzt werden konnten und weil die Trefferquote im Cache sich in der Praxis als sehr hoch erwiesen hat [RMS96].

4.7.3 Allokation der L-Register

Im letzten Kapitel haben wir ein noch recht einfaches Schema zur Allokation der L-Register beschrieben: dabei werden diese gleich nach Eintritt in eine Funktion alloziert und direkt vor dem Verlassen durch `return` wieder freigegeben. Hier sind verschiedene Varianten möglich, die von diesem strikten Schema abweichen.

Eine naheliegende Idee besteht darin, nicht einen einzigen L-Registersatz zu allozieren, sondern für den Rumpf einer Funktion mehrere solcher Registersätze bereitszustellen. So könnte man

für den Rumpf einer Funktion beispielsweise folgenden Code erzeugen, der 2 L-Registersätze verwendet:

```
allocate n
e1
deallocate
allocate m
e2
deallocate
```

Bei genauerer Betrachtung ergeben sich durch dieses Vorgehen allerdings keine Vorteile. Zum einen kann dadurch höchstens der Speicherverbrauch einer Funktion reduziert werden. Da die L-Register aber nach Verlassen einer Funktion wieder freigegeben werden, kann sich dadurch nur der maximale Speicherbrauch für L-Register in Abhängigkeit von der Rekursionstiefe verbessern. In absoluten Zahlen fällt dieser aber ohnehin nur sehr gering aus. Hier ist es dann günstiger, wenn man bei der Registerallokation L-Register doppelt belegt (vgl. Abschnitt 5.6.3). Darüber hinaus muß für diejenigen Variablen, die sowohl in e_1 als auch in e_2 verwandt wird, ein gewisser Mehraufwand in Kauf genommen werden: diese Variablen müssen von dem ersten Registersatz in den zweiten gerettet werden, was die Erzeugung von je einem `move`-Befehl vor dem ersten `deallocate` und nach dem zweiten `allocate` für jede dieser Variablen bedingt.

Eine Variation in der Verwendung der L-Register besteht darin, den `allocate` Befehl nicht gleich zu Anfang einzusetzen, sondern soweit nach hinten zu verschieben, bis zum ersten Mal eine Instruktion ein L-Register verwendet. Analog kann man `deallocate` nach vorne verlagern. Aber auch hier ergeben sich keine wirklichen Vorteile, da die Allokation ja ohnehin irgendwann stattfinden muß, man so den Aufwand höchstens verlagert aber nicht eingespart hat.

Durch Kombination der beiden obigen Techniken kann sich aber in der Praxis eine effektive Verbesserung ergeben: wir betrachten hierzu die Definition der Fibonacci-Funktion

```
fun fib(n) =
  case n<2 of
    true =>  1
  | false => fib(n-1)+fib(n-2)
```

Bei Eintritt in die Funktion befindet sich der Wert von `n` in $\mathtt{X}_0$. Der Test, ob `n` kleiner als 2 ist kann also vor der Allokation der L-Register bereits durchgeführt werden. Entsprechend kommt auch das erste Pattern ohne die Verwendung von L-Registern aus. Somit braucht für den ersten Fall überhaupt kein L-Registersatz alloziert werden. Betrachtet man sich den Aufrufbaum der Fibonacci Funktion, so erkennt man, daß dieser Fall sehr häufig (ungefähr in der Hälfte aller Fälle) auftritt. So macht sich die Einsparung der Allokation in der Praxis deutlich in der Laufzeit bemerkbar, da man ja nicht nur die Allokation selbst einspart, sondern auch das Retten und Laden von Variablen in diesen Bereich.

Die Idee der Optimierung besteht also darin, den `allocate` Befehl möglichst weit nach hinten zu verlagern. Erreicht man dabei ein Case, in dessen Wächter keine L-Register verwandt werden, dann dupliziert man den `allocate` Befehl und setzt das Verfahren für die einzelnen Zweige getrennt fort. Dieses Schema ist dem Vorgehen in der WAM sehr ähnlich: hier werden die L-Register (in der WAM heißen diese Y-Register) einer Funktion pro Klausel alloziert.

Viele rekursiv definierte Funktionen sehen so aus, daß der Basisfall in der Regel sehr einfach ist, ansonsten aber mehr Arbeit geleistet werden muß. Diese Optimierung zahlt sich besonders

Tabelle 4.2 L-Register: Anzahl der Registersätze pro Applikation und durchschnittliche Größe.

Anwendung	L-Registersätze pro Applikation	Größe der L-Registersätze
Prelude	46,45%	4,01
Browser	30,27%	3,93
Explorer	48,66%	7,84
Compile Browser	21,92%	3,71
Gump Scanner	47,28%	2,82
Gump Parser	34,26%	3,03
Scheduler (prove)	62,37%	4,46
Spedition	28,65%	3,29

bei sehr buschigen Rekursionsbäumen aus (wie im Beispiel der Fibonacci Funktion). Andererseits fällt etwa bei Listenrekursionen der Terminierungsfall nicht stark ins Gewicht, wenn man davon ausgeht, daß Listen in der Regel nicht leer sind. Dennoch zahlt sich die hier beschriebene Optimierung in der Praxis aus, wie Tabelle 4.2 belegt. So kann in vielen Fällen auf die Allokation eines L-Registersatzes völlig verzichtet werden und auch deren durchschnittliche Größe wird klein gehalten. Die Tabelle zeigt in der ersten Spalte, wieviele der Applikationen einer benutzerdefinierten Funktion zur Allokation eines L-Registersatzes führen. Man erkennt, daß dies nur in etwa höchstens der Hälfte aller Fälle nötig ist. Auch die durchschnittliche Größe der L-Registersätze fällt mit einem Durchschnitt von circa 4 Elementen über alle Anwendungen hinweg recht niedrig aus.

4.7.4 Programmtransformation

Für Oz stellt der Einsatz von Techniken aus der Programmtransformation im Zusammenhang mit Funktionen ein großes Potential für Optimierungen dar. Hier kann man versuchen, durch syntaktische Änderung des Quellprogramms zu einem semantisch äquivalenten, aber effizienteren Programm zu gelangen. Diese Techniken sind in der funktionalen Welt gut verstanden und können auch auf Oz (unter Berücksichtigung der Nebenläufigkeit) übertragen werden. Die Anwendung von Beta-Reduktion (inlining), Spezialisierung von Funktionen durch Falten und Auffalten, Herausfaktorisieren von Konstanten aus Funktionsrümpfen (hier vor allem geschachtelte Funktionsdefinitionen), etc. birgt sicher ein interessantes Optimierungspotential. Diese Techniken wurden in Mozart bisher noch nicht integriert, so daß Mozart in dieser Hinsicht hinter dem Stand der Kunst zurückbleibt.

Zusammenfassung

- *Codespezialisierung* ist eine allgemeine Optimierungstechnik, die besonders wichtig bei dynamisch getypten Sprachen ist, aber auch auf statisch getypte Sprachen anwendbar ist.

Dabei werden einzelne Instruktionen oder Sequenzen von Instruktionen durch speziellere Varianten ersetzt, da gewisse Eigenschaften der Eingaben für diese Instruktionen (wie z.B. Typkonsistenz) zugesichert werden können.

- Wir unterscheiden zwischen *statische* Codespezialisierung, die zur Übersetzungszeit durchgeführt wird und in *dynamische* Codespezialisierung, die zur Laufzeit durch selbstmodifizierenden Code erreicht wird.

- Codespezialisierung kann dann besonders effektiv für die formalen Parameter und die globalen Variablen einer Funktion f vorgenommen werden, wenn bei jeder Ausführung eine *Kopie des Rumpfes* von f angefertigt wird. In Mozart wird dies gerade für *Funktoren* (Funktionen die Module erzeugen) so gehandhabt.

- Aufbau von Tupeln und Zerlegung von Tupeln im Wächter kann durch Verwendung spezieller Register (Argument-Zeiger, Modus-Register) optimiert werden.

- Für Tupel in Form von Listen wird eine spezielle Darstellung gewählt, wodurch sowohl Speicher als auch Laufzeit gespart wird.

- Durch Codespezialisierung können etablierte Implementierungstechniken für Atome direkt auf Konstruktoren übertragen werden, so daß Konstruktoren in der Praxis genauso effizient gehandhabt werden können wie Atome.

- Bei vielen Instruktionen (wie z.B. Operatoren) ist eine *Spezialisierung* möglich, wenn bekannt ist, daß bestimmte Argumente bereits dereferenziert sind, oder ihr Typ oder gar ihr Wert bekannt ist. In der Praxis muß abgewogen werden, ob eine Hinzunahme von zusätzlichen spezialisierten Varianten von Instruktionen immer sinnvoll ist, da der Instruktionssatz möglich kompakt bleiben sollte.

- Viele Case-Ausdrücke haben eine einfache Struktur (z.B. nur eine Zahlkonstante im Wächter) und können daher effizienter übersetzt werden. In der Praxis können die meisten Case-Ausdrücke genauso effizient übersetzt werden wie vergleichbare Konstrukte in anderen Sprachen, trotz ihrer im allgemeinen komplexeren Semantik (Unifikation und Subsumption statt reines Matching).

- Große geschachtelte Case können durch *Indexing* Techniken effizient übersetzt werden. Ein Vorteil ergibt sich aber erst bei hinreichend großer Anzahl von Fällen.

- Die Optimierung von *Endrekursion* ist bei Sprachen ohne explizite Schleifen notwendig. Allerdings sind Schleifen im allgemeinen effizienter übersetzbar.

- Bei der Applikation einer Funktion f kann in den meisten Fällen eine Codespezialisierung vorgenommen werden, so daß Dereferenzierung, Typtest und Aritätsvergleich eingespart werden können. Zudem werden die zu erzeugenden Funktionsabschlüsse deutlich kleiner.

Kapitel 5

Diskussion

In den vorangehenden Kapiteln dieses Teils der Arbeit haben wir eine virtuelle Maschine für die Teilsprache L vorgestellt. Zugunsten der Übersichtlichkeit haben wir dabei bisher bewußt sowohl auf eine Diskussion der vorgestellten Techniken als auch auf die Besprechung von alternativen Ansätzen zur Implementierung verzichtet. Dies wollen wir nun in diesem Kapitel nachholen.

Wir beginnen mit einer Diskussion der Vor- und Nachteile einer stackbasierten gegenüber einer registerbasierten Maschinenarchitektur. Dann gehen wir im folgenden Abschnitt 5.2 auf die unkonventionelle Verwendung des Kellers in L ein. Darauf folgt in Abschnitt 5.3 eine Diskussion, wie Referenzketten entstehen können und wie man sie möglichst kurz halten kann. Danach gehen wir in Abschnitt 5.4 auf verschiedene alternative Möglichkeiten zur Darstellung von Funktionen ein. Es folgt in Abschnitt 5.5 eine kurze Diskussion zum Case. Anschließend gehen wir in Abschnitt 5.6 auf den Aspekt der Registerallokation in Zusammenhang mit L ein. Wir schließen das Kapitel ab, indem wir zeigen, wie durch eine einfache Erweiterung der Implementierung von logischen Variablen auch lazy Evaluierung unterstützt werden kann.

5.1 Stack- oder Registermaschine

Aus heutiger Sicht erscheint die Entscheidung für eine registerbasierte Architektur für die Oz-Maschine nicht unbedingt optimal. Die Wahl einer Registermaschine ist sicher dann sinnvoll, wenn die Erzeugung von nativem Code geplant ist, da diese Architektur der Hardwarearchitektur moderner RISC Maschinen sehr nahe kommt. Dadurch läßt sich dann eine sehr direkte und dadurch effiziente Abbildung der virtuellen auf die reale Maschine vornehmen.

Verwendet man hingegen zur Implementierung der virtuellen Maschine einen Emulator, so bietet eine stackbasierte Architektur, die auch von vielen virtuellen Maschinen für funktionale Sprachen verwandt wird [Lan64, Car83, PJ87], eine Reihe von Vorteilen. Bei einer Stackmaschine übernimmt der Keller zusätzlich auch die Funktion der Register: so wird der Keller auch für die Speicherung temporärer Variablen und für die Parameterübergabe verwandt. Die Register einer realen Maschine können sehr viel schneller zugegriffen werden als die Einträge des Kellers, der im Hauptspeicher liegt. Beim Emulator macht dies dagegen keinen Unterschied (bei wahlfreiem Zugriff auf die Einträge im Keller einer Stackmaschine), da hier auch die Register im Hauptspeicher liegen.

Bei einer Stackmaschine ist der Code in der Regel kürzer, da die Instruktionen mit weniger

Argumenten auskommen, weil der Keller hier als implizites Argument dient. So verknüpft beispielsweise eine Addition implizit die beiden obersten Elemente des Kellers und ersetzt diese beiden durch das Ergebnis. Diese Instruktion kommt also ganz ohne Argumente aus, während bei einer Registermaschine explizit die beiden Ein- und das Ausgabeargument angegeben werden müssen. Der Vorteil liegt hier nicht nur im reduzierten Speicherplatzverbrauch des Codes (und dadurch bedingten eventuellen Vorteilen hinsichtlich des Cacheverhaltens) sondern vor allem in einer erhöhten Ausführungsgeschwindigkeit der Instruktionen, da die Argumente beim Emulator erst aus dem Codebereich geladen und dekodiert werden müssen.

Stackmaschinen werden oft so entworfen, daß viele Instruktionen ein spezielles Akkumulatorregister als implizites Aus- und Eingabeargument verwenden. Der Akkumulator entspricht dabei in etwa einem gecachten obersten Element des Kellers. Das virtuelle Akkumulatorregister kann nun aber auch beim Emulator in der Regel auf ein reales Register der Hardware abgebildet werden, so daß in einer Stackmaschine hier wesentlich weniger Zugriffe auf den langsamen Hauptspeicher nötig werden als bei der Registermaschine.

Eine Stackmaschine bietet auch Vorteile bei der Parameterübergabe: die Registermaschine kopiert die Parameter zuerst in Register, von wo sie die gerufene Funktion in der Regel dann wieder auf ihren Keller bewegt. Dieser Schritt wird bei der Stackmaschine eingespart, da die Argumente ja direkt über den Keller übergeben werden.

5.2 Der Keller

Vergleicht man die Struktur des Kellers in L mit den etablierten Implementierungstechniken für rekursive Sprachen, so ergeben sich bei L deutliche Unterschiede. Dies betrifft zum einen das Vorgehen beim Funktionsaufruf und zum anderen die Darstellung der L-Register. Wir wollen beide Aspekte im folgenden näher diskutieren.

In Programmiersprachen, die rekursive Funktionen unterstützen, geht man in der Regel so vor, daß direkt beim Betreten einer Funktion ausreichend Speicher auf dem Keller alloziert wird, so daß dort sowohl die Rücksprungadresse für Funktionsaufrufe als auch die lokalen Variablen (in unserer Terminologie die L-Register) der Funktion gespeichert werden können. Dann muß bei jedem Funktionsaufruf nur noch der Programmzähler gerettet und beim Rücksprung restauriert werden. Beim Verlassen der Funktion wird dieses Kellersegment dann wieder freigegeben.

Wie in Kapitel 3 beschrieben, wird in L dagegen bei *jeder* (nicht endrekursiven) Funktionsapplikation ein neuer Auftrag auf dem Keller abgelegt, der die auf die Applikation folgende Instruktion enthält. Insbesondere muß also auch beim Rücksprung aus der gerufenen Funktion dieser Auftrag wieder vom Keller genommen werden.

Bei genauerer Betrachtung erkennt man aber, daß das gewählte Vorgehen durchaus nicht immer so ungünstig sein muß, wie es vielleicht auf den ersten Blick scheinen mag. Einerseits sind nämlich viele Aufrufe endrekursiv; in diesem Fall entstehen keine Mehrkosten. Da die Kelleroperationen in L nicht am Anfang einer Funktion sondern bei jeder Applikation stattfinden, treten erst dann Mehrkosten auf, wenn der Rumpf einer Funktion mehr als eine nicht-endrekursive Applikation einer benutzerdefinierten Funktion enthält. Durch das Oz-Schema erhöhen sich zwar die Kosten für einen Funktionsaufruf, allerdings beanspruchen die Kelleroperationen nur einen Teil an den Gesamtkosten für einen Funktionsaufruf. So dürfen etwa die Kosten für die Bereitstellung der Argumente nicht außer Acht gelassen werden.

Auch die Tatsache, daß die L-Register nicht direkt auf dem Keller alloziert werden, muß kein signifikanter Nachteil sein. Im Implementierungsteil in Abschnitt 7.1.2 werden wir sehen, daß die L-Register über Freispeicherlisten verwaltet werden, die fast genauso effizient gehandhabt werden können wie Kellerspeicher. Insbesondere ergibt sich hier auch kein Nachteil im Speicherverbrauch an sich. Durch das Vorgehen in L wird lediglich je ein zusätzlicher Allokations- und Deallokations-Schritt notwendig, Dieser fällt aber nicht sehr stark ins Gewicht, da jedes L-Register (wegen der Speicherbereinigung) auch noch bei der Allokation korrekt initialisiert werden muß (auch wenn man das konventionelle Schema verwenden würde). Zudem wird für die Lebensdauer eines L-Registersatzes auf jedes seiner Elemente unter Umständen mehrfach lesend und schreibend zugegriffen.

In Abschnitt 8.2 werden wir sehen, daß man den Zugriff auf den Keller des gerade in Ausführung befindlichen Threads optimieren kann, indem man den jeweils obersten Eintrag durch spezielle Maschinenregister cached. Dadurch muß man dann bei unserem Ansatz ein zusätzliches Register zur Adressierung lokaler Variablen in der Maschine vorsehen, was man im konventionellen Schema nicht braucht, da hier die L-Register direkt über das Top of Stack Register des Kellers zugegriffen werden können

Prinzipiell läßt sich das Standardimplementierungsschema des Kellers auch auf die in dieser Arbeit beschriebene Teilsprache L problemlos übertragen. Bei Mozart haben wir dennoch an dem unkonventionellen Vorgehen festgehalten: durch die sehr uniforme Struktur der Kellersegmente kann der Keller nämlich sehr einfach dazu benutzt werden, eine ganze Reihe unterschiedlichster Informationen zu verwalten (die allesamt effizient als Aufträge zur Codeausführung uniform dargestellt werden können, näheres siehe Kapitel 8). So gibt es beispielsweise spezielle Einträge zur Ausnahmebehandlung, für das Debugging, zur Realisierung der objektorientierten Erweiterungen der Sprache, die Implementierung von Locks oder zur Implementierung von lokalen Berechnungsräumen [Sch99]. Auch können hier mehrere Aufträge den selben L-Registersatz verwenden. Ein Rückgriff auf das konventionelle Schema hätte die Implementierung dieser Konzepte in der Regel deutlich erschwert oder gar ein völlig anderes Vorgehen verlangt.

Die WAM geht bei der Implementierung von Prozeduraufrufen noch einen Schritt weiter, indem die Rücksprungadresse nicht sofort auf dem Keller gespeichert wird, sondern zunächst in einem speziellen Register `CP` abgelegt wird. Dieses Vorgehen macht sich dann bezahlt, wenn die gerufene Prozedur selbst keine weiteren Prozeduren aufruft, da `CP` erst in diesem Fall auf den Keller gerettet werden muß. Prinzipiell ist dieses Vorgehen auch in Oz möglich, wurde aber in Mozart nicht realisiert, da es eine merkliche Erhöhung der Komplexität bedeutet hätte und zudem unklar ist, in wie vielen Fällen davon auch wirklich Gebrauch gemacht werden kann.

5.3 Referenzketten

Werden zwei Variablen miteinander unifiziert, so ist es zunächst freigestellt welche der beiden an die andere gebunden wird, also welche in einen `REF`-Knoten geändert wird. Obgleich solche Variable–Variable Bindungen in der Praxis recht selten auftreten (vgl. Tabelle 10.15 auf Seite 208), kann eine ungeschickte Wahl der Richtung der Bindung dennoch unangenehme Folgen haben: so traten in unserer Praxis vereinzelt Fälle auf, wo durch eine falsche Wahl der Bindungsrichtung sich die lineare Laufzeit eines Programmes in eine quadratische Laufzeit (in der Anzahl der verwandten Variablen) änderte.

Eine naheliegende Möglichkeit wäre, dem Benutzer Kontrolle über die Bindungsrichtung zu ge-

Tabelle 5.1 Länge von Referenzketten verschiedener Anwendungen.

Anwendung	Anzahl insgesamt	Ketten der Länge 0	1	2	> 2	Max. Länge
Laden Prelude	409.429	66,68%	32,37%	0,91%	0,04%	3
Gump Scanner	1.247.374	81,60%	17,69%	0,71%	0%	2
Gump Parser	23.593.547	85,06%	14,15%	0,72%	0,08%	18
Compile Browser	40.355.010	79,72%	19,13%	0,09%	0,01%	30
Browser	2.491.868	82,65%	17,27%	0,08%	0%	2
Explorer	3.103.079	60,51%	37,07%	2,42%	0%	2
Spedition	593.432	79,27%	20,30%	0,42%	0,01%	3
Scheduler (upper)	12.555.917	56,42%	43,29%	0,29%	0%	3
Scheduler (prove)	5.940.622	27,84%	68,50%	3,37%	0,29%	6
Durchschnitt	—	68,86%	29,97%	1,00%	0,04%	—

ben: so könnte man eine Unifikation `unify(`t`,`t'`)` eher als Zuweisung t`:=`t' auffassen und daher stets die Variablen von t präferiert binden. Dies würde allerdings dem Benutzer Entscheidungen auferlegen, die nur schwer mit dem hohen Abstraktionsniveau der Sprache zu vereinen sind, insbesondere da semantisch gesehen eine Gleichung der Form $t = t'$ völlig äquivalent zu $t' = t$ ist.

Zudem gibt es aus Sicht der Implementierung gute Kriterien, die in vielen Fällen eine bestimmte Bindungsrichtung besonders sinnvoll erscheinen lassen. Wird beispielsweise eine Variable x mit leerer Suspensionsliste mit einer Variable y unifiziert, deren Suspensionsliste nicht leer ist, so ist es günstiger, x an y zu binden, da dann keine Suspensionen geweckt werden müssen. Wie wir in Abschnitt 3.9 gesehen haben, ist es für die korrekte Implementierung von Case wichtig, daß lokale an globale Variablen gebunden werden. Treffen diese Kriterien nicht zu, so hat sich eine Heuristik als wirkungsvoll erwiesen: man bindet „jüngere“ präferiert an „ältere“ Variablen. Dabei wird das Alter einer Variablen über die Lage in der Halde entschieden (vgl. Abschnitt 7.1.1), also einen einfachen Vergleich von Zeigern. Somit werden neu erzeugte Variable an bereits vorhandene gebunden. Dieses Vorgehen kommt auch in Prolog zum Einsatz, ist hier allerdings keine Option sondern zwingend notwendig, damit beim Rücksetzen keine Referenzen in bereits wieder freigegebene Speicherbereiche entstehen.

Somit wird die Bindungsrichtung bei der Unifikation zweier Variablen nach folgenden Kriterien entschieden (mit absteigender Priorität):

1. Binde lokale an globale Variablen.
2. Binde Variablen mit leerer an solche mit nicht-leerer Suspensionsliste.
3. Binde jüngere an ältere Variablen.

Unter Verwendung dieser Regeln kann die Länge von Referenzketten in der Praxis fast in allen Fällen auf ein Minimum reduziert werden. Tabelle 5.1 zeigt vergleichend die Länge von Referenzketten bei verschiedenen Applikationen. Hier wurde die Anzahl der Aufrufe an die Funktion `deref` gezählt, und bei jedem Aufruf ermittelt, wieviele `REF`-Knoten jeweils betreten wurden.

Die letzte Zeile zeigt den Durchschnitt der Länge der Ketten über ihren prozentualen Anteil. Man erkennt, daß in mehr als zwei Dritteln aller Fälle beim Dereferenzieren überhaupt keine REF-Knoten betreten werden. Fast der gesammte Rest entfällt auf einstufige Ketten: diese entstehen, wenn eine Variable an einen Wert gebunden wird, der keine Variable ist. Das entspricht dem üblichen Programmierschema in Mozart, wo Funktionen ihre Rückgabewerte über Variablen an die rufende Funktion übergeben. Für eine funktionale Sprache wie L kann man daher mit einem deutlich geringeren Anteil an Referenzketten rechnen. Weniger als 1% aller Ketten haben die Länge 2, was bei der Unifikation zweier Variablen entsteht. Dies wird vor allem durch ein Manko des Compilers bedingt, der bestimmte verschachtelte Ausdrücke suboptimal auffaltet und so unnötigerweise zwei statt einer Variablen erzeugt.

Ketten, die länger als 2 sind, treten in 5 Applikationen auf, und kommen hier in maximal jedem 1250. Aufruf an die Funktion `deref` vor. Dennoch können die Ketten beispielsweise beim Compiler mit einer Länge von bis zu 30 Knoten eine recht imposante Länge annehmen, obgleich diese in der Praxis wegen ihres verschwindend geringen Vorkommens keinen signifikanten Einfluß haben kann. Eine genauere Untersuchung, wie es dennoch zu solch langen Ketten kommen kann, zeigte, daß es hier zu einem ungünstigen Zusammenspiel zwischen Differenzlisten einerseits und tief verschachtelten Termen andererseits kam. Dies läßt sich gut an einem einfachen Beispiel verdeutlichen: die Funktion `flatten` transformiert eine beliebig verschachtelte Listenstruktur in eine flache Darstellung unter der Verwendung von Differenzlisten, also Listen deren Ende eine ungebundene Variable bildet, so daß sich daran sehr effizient neue Elemente anhängen lassen:

```
fun flatten(l,hd,tl)
  case l of
    h::r => let val aux=lvar()
            in flatten(h,hd,aux);
               flatten(r,aux,tl)
            end
  | nil  => unify(hd,tl)
  | _    => unify(hd,l::tl)
```

`l` ist hier die zu verflachende Eingabestruktur, `hd` das Ende der bereits berechneten Teillösung und `tl` das Ende der kompletten Ausgabeliste.

Der folgende Ausdruck

```
let val hd=lvar() in flatten([[1]],hd,nil) end
```

wird dann wie folgt ausgeführt:

```
flatten([[1]],hd,nil)
flatten([1],hd,aux1)                          flatten(nil,aux1,nil)
flatten(1,hd,aux2)    flatten(nil,aux2,aux1)  flatten(nil,aux1,nil)
hd=1::aux2            flatten(nil,aux2,aux1)  flatten(nil,aux1,nil)
hd=1::aux2            aux2=aux1               flatten(nil,aux1,nil)
hd=1::aux2 aux2=aux1  aux1=nil  % aux2 ⟶ aux1 ⟶ nil
```

Zunächst wird hier die Variable `aux2` an `aux1` gebunden, weil sie jünger ist. Danach wird dann `aux1` an `nil` gebunden. Durch Erhöhung der Schachtelungstiefe der Eingabe lassen sich so belie-

big lange Referenzketten erzeugen. Man beachte, daß die anfangs diskutierte präferierte Bindung der Variablen der linken Seite im Sinne einer Zuweisung `hd:=tl` keine Änderung bedingt.

Die Resultate aus Tabelle 5.1 zeigen, daß eine mögliche Optimierung wenig erfolgversprechend ist: es macht keinen Sinn, nach dem Dereferenzieren am Ausgangspunkt das Ende der Kette zu speichern, da diese ohnehin in der Regel sehr kurz sind. Hier majorisiert der schreibende Speicherzugriff in der Regel die Einsparung beim lesenden Zugriff. Zudem werden viele Werte nur ein einziges Mal zugegriffen. Für häufiger verwandte Werte bietet die Speicherbereinigung eine wesentlich bessere Optimierungsmöglichkeit: die meisten Referenzketten können hier entweder völlig eliminiert (wenn sie nicht in einer Variable münden) oder zumindest auf die Länge 1 reduziert werden.

5.4 Funktionen

Im folgenden diskutieren wir in Abschnitt 5.4.1 alternative Möglichkeiten zur Darstellung von Funktionen und besprechen in Abschnitt 5.4.2 eine Möglichkeit, die es erlaubt, auf die G-Adressierungsart zu verzichten.

5.4.1 Darstellung der Abschlüsse

Wie in Abschnitt 3.3 beschrieben verwenden wir bei der Darstellung von Funktionen ein flaches Schema zur Repräsentation der globalen Variablen, das heißt alle globalen Variablen einer Funktion werden in *einem* G-Registersatz alloziert.

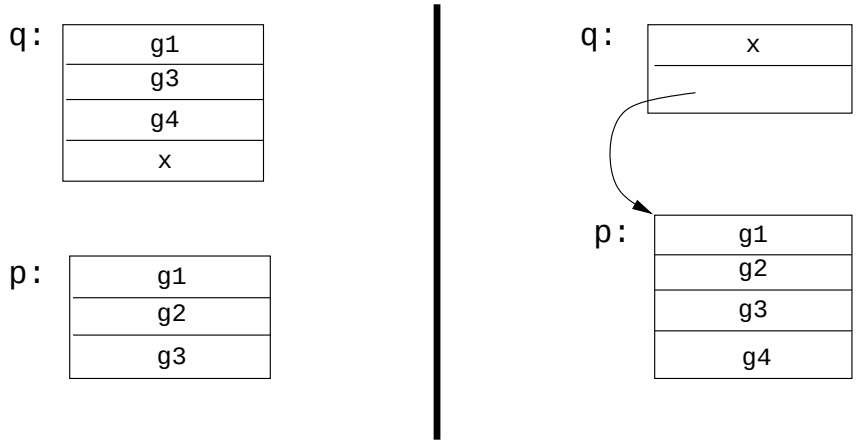
In Algol-ähnlichen Sprachen werden üblicherweise die Abschlüsse von Funktionen kaskadiert aufgebaut, entsprechend der Schachtelungstiefe der Funktion. Wir erläutern das Vorgehen am Beispiel:

```
let
  val g1=...; val g2=...; val g3=...; val g4=...
  fun p(x) =
    let fun q(y) = g1+g3+g4+x+y
    in q(g1+g2+g1+g3) end
    end
  in ... end
end
```

Hier handelt es sich um zwei Funktionen `p`, die die globalen Variablen `g1`, `g2` und `g3` verwendet und `q` mit den globalen Variablen `g1`, `g3`, `g4` und `x`. Abbildung 5.1 zeigt links die Darstellung der G-Register in Mozart. Der rechte Teil der Abbildung zeigt eine alternative Repräsentation für die G-Register: der G-Registersatz der inneren Funktion `q` hat hier auch noch eine Referenz auf den G-Registersatz der sie umgebenden Funktion `p`. Die G-Register von `q` enthalten hier nur diejenigen Variablen, die global zu `q` sind, aber nicht diejenigen, die auch global zu `p` sind. Somit enthalten die G-Register von `q` eine Referenz auf die von `p` und somit nur den Wert der Variablen `x`.

Bei tiefer Schachtelungstiefe von Funktionen und häufigem Zugriff auf weiter entfernte Variablen liegt hier der Vorteil in einer durchschnittlichen Verringerung der Größe der Abschlüsse

Abbildung 5.1 Unterschiedliche Darstellung der G-Register



und damit verbunden einer Reduzierung der Kosten beim Aufbau. Allerdings wird dadurch der Zugriff auf eine globale Variable in Abhängigkeit von ihrer Entfernung teurer; inbesondere wird die Adressierung globaler Variablen komplexer und gerade in einem Emulator sehr teuer, da hier sehr feinkörnige Instruktionen definiert werden müssen. In diesem Schema hat eine innere Funktion auch stets Referenzen auf alle globalen Variablen, der sie umgebenden Funktionen, was sich bei der Speicherbereinigung als nachteilig erweisen kann, da so nicht nur die tatsächlich verwendeten Werte kopiert werden müssen.

Dieses Vorgehen schien für Mozart daher nicht sinnvoll, da, wie Tabelle 10.13 auf Seite 206 belegt, die durchschnittliche Größe von Umgebungen ohnehin sehr klein ausfällt. Zudem trifft man in Programmen selten eine Schachtelungstiefe größer als 2 an.

Shao und Appel [SA94] schlagen ein noch weiter elaboriertes Verfahren vor, bei dem Funktionen nicht nur die G-Register der sie direkt umgebenden Funktion referieren können. Hier wird vielmehr versucht, einen optimalen Graphen von verzeigerten G-Registersätzen aufzubauen, so daß möglichst wenig Redundanz entsteht. Shao und Appel berichten von deutlichen Einsparungen in Speicherplatz und Laufzeit für große SML-Programme, was allerdings dadurch bedingt wird, daß die dort verwandte Technik des CPS (= continuation passing style) [App92] sehr viele Funktionen dynamisch erzeugt, so daß deren Ergebnisse nicht ohne weiteres auf Oz übertragbar sind.

5.4.2 Wegfall der G-Adressierung

Ein alternatives Schema zur Behandlung globaler Variablen kann auch wie folgt aussehen: hat eine n-stellige Funktion k globale Variablen, so werden diese beim Aufruf wie zusätzliche Argumente in den Registern n bis $n + k - 1$ übergeben. Die Darstellung einer Funktion selbst ändert sich dadurch nicht: beim Ausführen der Funktiondeklaration muß dann zwar wie bisher eine Funktion erzeugt werden, der Rumpf der Funktion kann aber ohne Kenntnis des Konzeptes der globalen Variable übersetzt werden. Der Rumpf der Funktion `f` mit zwei globalen Variablen `g1` und `g2`

```
fun f(x) = g1+g2
```

wird dann also so übersetzt, als wäre sie wie folgt definiert worden:

```
fun f(x,g1,g2) = g1+g2
```

Der Vorteil dieses Vorgehens liegt in einer Vereinfachung des Compilers und der Maschine durch den Wegfall einer Adressierungsart. Auf der anderen Seite wird dadurch die Applikation durch das Kopieren der globalen Variablen in die X-Register teurer. Zudem müssen diese nun zu lokalen Variablen mutierten globalen Variablen in der Regel in L-Registern gesichert werden, was zusätzliche Laufzeitkosten und vergrößerte L-Registersätze bedeutet.

5.5 Case

In Abschnitt 3.9 haben wir die Implementierung von Case so beschrieben, daß im Falle der Suspension der suspendierende Wächter nach dem Wecken komplett noch einmal von vorne ausgeführt wird. Dieses Vorgehen ist allerdings nicht immer optimal. Betrachten wir das Beispiel Beispiel

```
case x of
  [1] => ...
| _   => ...
```

und nehmen an, daß `x` bereits an eine Liste der Form `[y]` mit einer logischen Variable `y` gebunden ist. Dann wird zunächst auf `y` suspendiert. Wenn `y` dann später gebunden wird, wird der Wächter noch einmal vollständig neu ausgeführt, insbesondere wird noch einmal unnötigerweise getestet, ob `x` an eine einelementige Liste gebunden ist, obwohl ein Vergleich von `y` mit `1` ausreichend ist. Dieser zusätzliche Test ist allerdings billig, so daß ein aufwendigeres Implementierungsschema, wie wir es im folgenden diskutieren werden, in diesem konkreten Fall keinen Gewinn bringen wird. In anderen Fällen kann sich das bisher besprochene Schema allerdings als nachteilig erweisen; wenn nämlich beim Gleichheitstest von `x` und `y` mittels folgendem Ausdruck

```
case [x,y] of
  [z,z] => ...
| _     => ...
```

beispielsweise sowohl `x` als auch `y` an sehr lange offene Listen (das heißt das Ende jeder Liste ist jeweils eine logische Variable) gebunden sind. Wenn nun diese Listen sukzessive am Ende erweitert aber nicht geschlossen werden, dann wird bei jeder Erweiterung das Case geweckt und die Unifikation von `x` und `y` durchgeführt, das heißt jedesmal werden beide Listen komplett von vorne durchlaufen, obwohl eigentlich nur die neuen Enden unifiziert werden müßten.

Dieses Problem läßt sich durch ein alternatives Implementierungsschema beseitigen [ST94]: wie bereits in Abschnitt 3.9 beschrieben, enthält die Spur bei Erreichen des Wächters eine Liste von Gleichungen der Form $x_1 = t_1, \ldots, x_n = t_n$, wobei dann auf alle x_i suspendiert wird. Man kann nun so vorgehen, daß man sich beim Suspendieren diese Gleichungen explizit an der erzeugten Suspension merkt; dies kann effizient geschehen, indem man sich nur jeweils zwei Referenzen

(auf x_i und t_i) merken muß und nicht etwa eine Kopie der t_i zu erzeugen ist. Beim Wecken wird dann der Wächter überhaupt nicht mehr ausgeführt; vielmehr werden für die zuvor gemerkten Gleichungen noch einmal die linken und rechten Seiten unifiziert. Schlägt eine dieser Unifikationen fehl, so wird zum nächsten Wächter gegangen. Ist die Spur nach diesen Unifikationen leer, so liegt Subsumption vor. Ist die Spur nicht leer, so wird wie zuvor auf die nun darin enthaltenen neuen Gleichungen suspendiert.

Bei der Implementierung von Mozart kommen beide Schemata zum Einsatz. Bei Konditionalen, die nur Gleichungen enthalten, kommt in Mozart genau das zuvor beschriebene Schema zum Einsatz. Hier haben sich die oben erwähnten Nachteile in der Praxis als nicht signifikant erwiesen, weswegen wir das einfachere Schema vorgezogen haben.

Gegenüber L unterstützt Oz aber auch *tiefe* Wächter, das heißt ein Wächter kann beliebige Ausdrücke der Sprache enthalten. Daher ist es auch in der Regel weder effizient noch korrekt (wegen Seiteneffekten) einen solchen Wächter nach Suspension neu auszuführen. Wenn ein Wächter mehr als nur Gleichungen enthält, wird daher hierfür in Mozart ein völlig anderes Implementierungsschema verwandt, das auf der oben beschriebenen Idee basiert [MSS95, Meh99].

5.6 Registerallokation

Bei der Registerallokation handelt es sich um eine klassische Aufgabenstellung aus dem Compilerbau. Wir wollen an dieser Stelle auf einige spezielle Aspekte im Zusammenhang mit L näher eingehen.

Die Aufgabe der Registerallokation besteht darin, für jede Variable x eines Programmes eine Adressierungsart (X, L, G oder V) zu bestimmen und x auf einen Index i des zugehörigen Registersatzes abzubilden, wobei man am günstigsten funktionsweise vorgeht.

5.6.1 Zuordnung der Adressierungsart

Zunächst ist es einfach, zu bestimmen, welche Variablen auf V- und G-Register zu verteilen sind: alle Toplevel Variablen werden V-Registern zugeordnet. Alle anderen freien Variablen einer Funktion gehören in G-Register. Im Implementierungsteil werden wir sehen, daß manche Instruktionen als Argumente keine V-Register zulassen, so müssen dann unter Umständen auch Toplevel Variablen auf G-Register abgebildet werden, wenn sie in solchen Instruktionen verwandt werden. Für G- und V-Register ist es prinzipiell egal, auf welchen Index eine Variable abgebildet wird, wenn man davon ausgeht, daß auf jeden Index gleich effizient zugegriffen werden kann.

Alle verbleibenden Variablen, also gerade die lokalen Variablen einer Funktion einschließlich der formalen Parameter, kommen in X- und L-Register. Hier hängt die Qualität der Registerallokation von verschiedenen Faktoren ab: zum einen sollten möglichst viele Variablen auf X-Register abgebildet werden, da diese anders als die L-Register nicht dynamisch verwaltet werden müssen. Zudem kommt es entscheidend darauf an, welcher Variablen welcher Index zugeordnet wird: zum einen ist es günstig, mit möglichst wenig Registern auskommen, zum anderen lassen sich durch geschickte Verteilung Instruktionen einsparen.

5.6.2 Temporäre und permanente Variablen

Je nachdem ob man eine Variable ein X- oder ein L-Register zuordnet bezeichnet man diese auch als *temporär* oder respektive *permanent*.

Die Klassifikation in permanent oder temporär hängt von der Lebensdauer einer Variable ab: wenn ein Berechnungspfad existiert, auf dem eine Variable sowohl vor als auch nach einem Funktionsaufruf auftritt, dann wird sie als permanent klassifiziert alle anderen lokalen Variablen sind temporär. Grund hierfür ist die Tatsache, daß durch einen Funktionsaufruf in unserem Übersetzungsschema die X-Register von der gerufenen Funktion beliebig verändert werden können.[1] Der Compiler kann also im allgemeinen nicht davon ausgehen, daß eine Variable, die sich vor dem Aufruf in einem bestimmten X-Register befand, nach dem Aufruf noch über dieses Register zugreifbar ist.

5.6.3 L-Register

Bei der Zuordnung von Indizes für die permanenten Variablen können verschiedene Optimierungskriterien angewandt werden. Zunächst kann man versuchen, mit möglichst wenigen Indizes auszukommen, das heißt man versucht den L-Registersatz möglichst klein zu halten. Dies kann beispielsweise erreicht werden, indem man Variablen, deren Lebensdauer sich nicht überlappt, auf einen gemeinsamen Index abbildet. Der damit verbundene Gewinn an Speicherplatz ist dabei allerdings unbedeutend: die L-Registersätze werden ja in einer Stackdisziplin verwaltet (auch wenn sie nicht direkt auf dem Keller alloziert werden), wobei die Rekursionstiefe bedingt durch Endrekursionsoptimierung im allgemeinen recht niedrig ist. Nach Verlassen einer Funktion werden die L-Register wieder freigegeben und können wiederverwandt werden. Das heißt, durch Verringerung der Größen einzelner L-Registersätze läßt sich nur der maximale Speicherverbrauch einer Applikation reduzieren. Wie Tabelle 5.2 bestätigt, benötigt der Keller eines Threads maximal Speicher in der Größenordnung einiger weniger Kilobytes, so daß im Hinblick auf den Gesamtspeicherplatzverbrauch dieser Applikationen nur mit einem vernachlässigbaren Gewinn an Speicherplatz gerechnet werden kann.

Ein wichtigeres Optimierungskriterium kommt dagegen im Zusammenhang mit der Speicherbereinigung zum tragen. Dabei müssen natürlich auch Threads und deren Aufträge und damit wiederum die davon referierten L-Registersätze bearbeitet werden. Nun kann ein L-Registersatz zum Zeitpunkt der Speicherbereinigung noch Referenzen auf Knoten im Speicher enthalten, obwohl die zugehörigen Register im Code gar nicht mehr verwendet werden. Da im allgemeinen aber nicht zu erkennen ist, welche Register noch benötigt werden, wird man hier konservativ alle Register als lebendig behandeln und so (unter Umständen große) Speicherbereiche kopieren, obwohl dies gar nicht nötig wäre.

Eine Lösung kann darin bestehen, daß man den Punkt von oben noch einmal aufgreift und versucht, die L-Registersätze möglichst klein zu halten, da mit kleineren Registersätzen die Wahrscheinlichkeit sinkt, daß man unnötige Datenstrukturen kopiert. Dennoch ist die Lösung nicht immer optimal: oft kann das L-Register einer nicht mehr benötigten Variablen nicht wiederverwandt werden, da es keine andere Variable mit geeigneter Lebensspanne gibt. In anderen Fällen kann es auch passieren, daß nach der letzten Verwendung eines Registers und vor seiner Wieder-

[1] Ein anderes gängiges Übersetzungsschema (callee save) sieht so aus, daß eine Funktion diejenigen X-Register, die sie ändert, zuvor in lokale L-Register rettet, so daß sie vor Verlassen der Funktion wieder restauriert werden können.

Tabelle 5.2 Maximale Tiefe des Kellers in Byte für verschiedene Applikationen

Anwendung	Max. Tiefe des Kellers in Byte
Prelude	816
Browser	1.176
Explorer	4.692
Compile Browser	30.864
Gump Scanner	2.280
Gump Parser	15.804
Scheduler (upper)	1.452
Scheduler (prove)	1.404
Spedition	444

verwendung eine Speicherbereinigung stattfindet.

Eine optimale Lösung hingegen könnte so aussehen, daß der Compiler nach der jeweils letzten Verwendung einer permanenten Variablen das entsprechende Register als unbenutzt markiert. Das bedeutet allerdings merkliche Einbussen bei der Laufzeit von Programmen, die man nicht gerne in Kauf nehmen will, insbesondere wenn man bedenkt daß diese Information ja nur sehr selten auch aktiv genutzt wird.

Eine weitere optimale Lösung kann realisiert werden, wenn man sich an den Stellen, an denen ein Aufruf der Speicherbereinigung prinzipiell stattfinden könnte, merkt, welche L-Register noch lebendige Werte enthalten. In L sind diese Stellen gerade die Funktionsapplikationen. Man könnte also die Instruktionen zur Funktionsapplikation (`apply`, `tailApply`, etc.) um ein zusätzliches Argument erweitern, das Aufschluß über die lebendigen L-Register gibt. Dieses Argument würde bei der normalen Ausführung der Instruktionen ignoriert und nur bei der Speicherbereinigung berücksichtigt, so daß sich keine Effizienzeinbußen ergeben würden.

In DFKI Oz 2 haben wir keine der oben besprochenen Optimierungen vorgenommen; in Mozart dagegen werden Variablen, deren Lebensdauer sich nicht überlappt, möglicherweise auf das gleiche L-Register abgebildet.

Die WAM verwendet in dieser Hinsicht ein optimales Schema: L-Registersätze (die in der WAM Y-Register heißen) werden dort pro Klausel direkt auf dem Keller alloziert. Für diese wird dann die Technik des *environment trimming* verwandt. Dabei werden die lokalen Variablen einer Funktion so auf die L-Register verteilt, daß Variablen, deren Lebensdauer eher endet, weiter oben auf dem Keller zu liegen kommen. Bei jedem Funktionsaufruf kann dann das Kellersegment der rufenden Funktion um die nicht mehr benötigten Variablen verkleinert werden. Da eine Klausel selbst nun keine Verzweigungen etwa in Form von Konditionalen mehr enthalten kann, enthält der Keller so immer genau die noch lebendigen Variablen. Environment trimming ist in L nicht möglich, da die L-Register hier ja nicht auf dem Keller alloziert werden.

5.6.4 X-Register

Bei der Allokation der X-Register handelt es sich um das eigentliche Registerallokationsproblem im klassischen Sinne des Compilerbaus [AVA86]. Auch hier kann man nach verschiedenen Kriterien optimieren. Zum einen kann man versuchen, die Anzahl der verwandten X-Register zu minimieren. Das ist besonders bei nativer Codeerzeugung wichtig, da hier die Anzahl der Register, die die Hardware zur Verfügung stellt, in der Regel klein ist. Bei einem Emulator ist dieses Kriterium allerdings unwichtig, da hier aus technischer Sicht problemlos sehr viele Register zur Verfügung gestellt werden können; das gilt inbesondere deshalb, da es, wie in Kapitel 8 beschrieben, in der Implementierung erreicht werden kann, daß X-Register nicht per Thread sondern einmal für die ganze Maschine alloziert werden.

Ein anderes Kriterium versucht die Anzahl der verwendeten `move` Befehle zu minimieren, was gerade beim Emulator von Vorteil ist, da hier der interpretative Aufwand für diese Instruktion im Vergleich zum Nutzen besonders hoch ist. Man geht dann so vor, daß man die konkrete Zuordnung temporärer Variablen zu den X-Registern noch offen läßt und Maschinencode erzeugt, der Referenzen auf diese Variablen noch in symbolischer Form enthält. Ist der Compiler in einer Programmiersprache mit logischen Variablen geschrieben, so läßt sich das besonders elegant lösen, indem man logische Variablen für die einzelnen noch zu bestimmenden Indizes vorsieht.

Die Berechnung einer optimalen Zuordnung wird in der Regel als Graphenfärbungsproblem formuliert und kodiert, das heißt das Problem selbst ist im allgemeinen NP-hart. Daher wurde bei der Implementierung des Compilers für DFKI Oz ein anderes Vorgehen gewählt. Hier wurde nicht versucht, die Registerzuordnung auf der Ebene der virtuellen Maschineninstruktionen vorzunehmen, sondern bereits früher auf der Ebene der abstrakten Syntax angesetzt. Dem lag die Annahme zugrunde, daß auf diesem höheren Level zum einen leichter zu erkennen ist, welche Zuordnung optimal ist und zum anderen die Algorithmen einfacher zu definieren sind. Dieses Vorgehen hat sich allerdings im Nachhinein als nachteilig herausgestellt: zwar ist die Qualität der erzeugten Registerzuordnung durchaus gut, allerdings mußte zur Sicherstellung der Korrektheit immer wieder nachgebessert werden. Die Ebene der virtuellen Maschineninstruktionen hat nämlich den Vorteil, daß diese sehr einfach und regulär ist, so daß die Korrektheit der verwandten Algorithmen einfacher sichergestellt werden kann. Dagegen ist die Schicht der abstrakten Syntax zu komplex für diese Aufgabe, so daß sich hier leicht Fehler einschleichen konnten, deren Auffinden und Beseitigung einen nicht unerheblichen Umfang angenommen hat. Auch bedingten häufig vorgenommene Erweiterungen des Compilers (was bei einem stark in Entwicklung befindlichen System vor allem wegen Hinzunahmen von neuen und Änderungen an bestehenden Sprachkonstrukten häufig vorkam) fast immer eine fehlerträchtige Einbeziehung der Registerallokation, was beim klassischen Ansatz sicher seltener der Fall gewesen wäre.

Ein anderer interessanter Lösungsansatz, der bisher noch nicht weiter verfolgt wurde, wäre der, das Problem der optimalen Registerallokation als Constraintproblem über endlichen Bereichen [Wür98] zu formulieren. Hier wäre interessant herauszufinden, wie sich dieses Vorgehen sowohl auf die Laufzeit als auch auf den Speicherverbrauch des Compilers auswirken würde und wie die generierten Lösungen sich zu den suboptimalen (aber in der Praxis doch recht guten) Ergebnissen der aktuellen Implementierung verhalten.

5.7 Lazy Evaluierung

Wir wollen in diesem Abschnitt kurz auf lazy Evaluierung eingehen, die aus Implementierungssicht eng mit dem der logischen Variable verknüpft ist.

Bei der lazy Evaluierung wird bei der Deklaration einer Variable `x`

```
let val x = e in ... end
```

der Wert von `x` nicht sofort durch die Auswertung von e bestimmt. Vielmehr wird zunächst bei der Definition mit `x` nur eine nullstellige Funktion `fn()=>`e assoziiert. Erst wenn der Wert von `x` benötigt wird, wird diese Funktion aufgerufen und `x` entsprechend ersetzt.

Dieses Verhalten läßt sich zwar mit logischen Variablen indirekt kodieren [Iba95], hat aber den Nachteil, daß ein Zugriff auf solche Variablen nicht mehr transparent erfolgen kann. Durch eine kleine Erweiterung der Implementierung logischer Variablen können aber solche lazy Variablen direkt und damit transparent von der Maschine unterstützt werden.

Lazy Variablen werden als Unterklasse von logischen Variablen definiert, die mit zwei zusätzlichen Attributen ausgestattet sind: eines ist eine Referenz auf die Funktion zur Berechnung des Wertes, beim zweiten handelt es sich um eine boolesche Variable, die anzeigt, ob die Berechnung des Wertes bereits angestoßen wurde (dies wird wegen der Nebenläufigkeit benötigt). Lazy Variablen werden dann innerhalb der Maschine überall wie logische Variablen behandelt, lediglich wenn der Wert einer solchen Variable benötigt wird, muß eine Sonderbehandlung durchgeführt werden. Die Stellen, die anzeigen, wann der Wert eine Variablen `x` benötigt wird, sind nun nicht etwa über die ganze Maschine verteilt, es gibt vielmehr nur zwei genau definierte Stellen, an denen ein Eingriff nötig wird. Das ist zum einen die Unifikation (oder genauer das Binden) einer lazy Variablen und zum zweiten die Stelle, an der eine Suspension zu einer Variablen hinzugefügt wird. In beiden Fällen wird dann, wenn noch nicht geschehen, die Berechnung des Wertes von `x` angestoßen, an dessen Ende `x` dann gebunden wird.

In Oz werden Ausdrücke strikt ausgewertet, die Verwendung von lazy Variablen wird durch explizite Deklaration erlaubt. Die Implementierung von lazy Variablen in Mozart ist derzeit auf eine grobkörnige Verwendung angelegt. So werden diese beispielsweise vorwiegend zum verzögerten Laden von Modulen verwandt; erst wenn zum ersten Mal auf ein bestimmtes Modul zugegriffen wird, wird dieses geladen.

Zusammenfassung

- Eine *Stackarchitektur* beim Design der virtuellen Maschine bietet gegenüber der für L gewählten Registermaschine Vorteile, wenn man als Implementierungstechnik einen Emulator verwendet: viele Argumente der Instruktionen sind implizit und ein Akkumulator-Register kann auf ein natives Register abgebildet werden, so daß sich der interpretativen Aufwand reduziert.
- Der *Keller* von L hat eine unkonventionelle Struktur: er enthält nur gleichgroße Aufträge (dargestellt als Tripel $\langle$`PC`, `L`, `G`$\rangle$), während die L-Register auf der Halde alloziert aber den-

noch über eine Freispeicherliste in einer Stackdisziplin verwaltet werden. Dieses Vorgehen bedeutet etwas erhöhte Kosten beim Funktionsaufruf, hat aber den Vorteil, daß auf dem Stack unterschiedlichste Informationen (Debugging, Ausnahmebehandlung, Profiling, tiefe Berechnung, etc.) abgelegt werden können.

- *Referenzketten* stellen in der Praxis von Mozart kein Problem dar: zwei Drittel aller Dereferenzierungsschritte finden direkt eine Zelle vor, die nicht mit `REF` markiert ist. Fast das gesammte verbleibende Drittel sind Referenzketten der Länge 1. Diese Zahl wäre sicherlich noch deutlich geringer, wenn in Mozart für die Ausgabe einer Funktion nicht jeweils eine logische Variable erzeugt werden müßte.

- Alternativen für die Darstellung der *Abschlüsse* von Funktionen sind denkbar, wie etwa eine kaskadierte Darstellung mit Referenzen auf den Abschluß der direkt umgebenden Funktion.

- Bei Suspension und späterem Wecken eines Case wird der suspendierte Wächter stets noch einmal von vorne ausgeführt. In seltenen Fällen könnte hier ein Vorgehen wie bei tiefen Wächtern sinnvoller sein: beim Suspendieren merkt man sich nur diejenigen Gleichungen, für die Subsumption noch nicht entschieden werden konnte.

- Für alle Variablen wird je nach ihrem Vorkommen eine Adressierungsart (X, L oder G) bestimmt. Danach wird eine Zuordnung auf Registerindizes vorgenommen. Für die X-Register ergibt sich das klassische Problem der *Registerallokation*.

- *Lazy Variablen* lassen sich einfach durch eine Erweiterung logischer Variablen implementieren.

Teil III

Implementierung

Kapitel 6

Der Emulator

Nachdem wir in Teil II ein Modell der virtuellen Maschine für die Sprache L vorgestellt haben, wollen wir uns in diesem Teil der Arbeit in einem weiteren Verfeinerungsschritt den Aspekten der Implementierung dieses Modells zuwenden. Wir gehen dabei von den Erfahrungen und Erkenntnissen aus, die wir bei der konkreten Implementierung von DFKI Oz und dessen Nachfolgesystem Mozart gewonnen haben.

Wir werden in diesem Teil der Arbeit zeigen, wie gegenüber dem abstrakten und idealisierten Modell der Maschine aus Teil II aus pragmatischer Sicht gewisse Einschränkungen und Zugeständnisse nötig werden. Beispielsweise werden wir in Kapitel 8 sehen, daß es günstiger ist, die X-Register nicht für jeden Thread separat zu allozieren, sondern einmalig für die ganze Maschine.

Weiter werden wir in diesem Teil auf Aspekte eingehen, die in Teil II noch unberücksichtigt blieben, aber aus praktischer Sicht von großer Bedeutung sind, wie etwa die genaue Darstellung von Werten im Speicher (siehe Kapitel 7).

Die Implementierung von Mozart verwendet, wie bereits erwähnt, nicht den Ansatz der nativen Codeerzeugung, sondern implementiert das Maschinenmodell unter Verwendung eines Emulators in C++. Daher werden wir in diesem Teil eine Reihe von Implementierungstechniken beleuchten, die speziell im Zusammenhang mit Emulatoren eingesetzt werden können.

Wir werden uns in diesem Kapitel zunächst auf den Kern des Emulators konzentrieren und werden anschließend in Kapitel 7 das gesammte Speichermanagement des Emulators beschreiben. Darauf folgt in Kapitel 8 eine Beschreibung der Implementierung von Threads. Daran anschließend gehen wir in Kapitel 9 darauf ein, wie man primitive Funktionen (die nicht in Oz implementiert sind) in Oz verfügbar machen kann. Wir schließen diesen Teil dann mit einer Diskussion über die Implementierung des Case ab.

In diesem Kapitel geben wir zunächst im folgenden Abschnitt einen kurzen Überblick über die Register und Datenbereiche des Emulators. Danach folgt in Abschnitt 6.2 eine Beschreibung der Repräsentation des Maschinencodes. Es folgt eine Beschreibung des Interpreters in Abschnitt 6.5. Dem schließt sich die Beschreibung von threaded code an, einer wichtigen Implementierungstechnik für Emulatoren. In Abschnitt 6.7 diskutieren wir die Vor- und Nachteile der nativen Codeerzeugung. Wir schließen das Kapitel mit der Beschreibung einer Reihe weiterer Implementierungstechniken, die speziell auf Emulatoren anwendbar sind.

6.1 Überblick

In diesem Abschnitt wollen wir einen Überblick über die bei der Implementierung der Maschine verwendeten Register und die damit verbundenen Datenbereiche geben. Diese stimmen nämlich nicht völlig mit dem in Teil II beschriebenen Modell überein. So werden etwa in der Implementierung die X-Register nicht pro Thread sondern einmal für die ganze Maschine verwaltet. Auch verwendet die Implementierung noch zusätzliche Register, die nicht in Teil II benötigt wurden, da die Maschine zum Beispiel den obersten Auftrag eines jeden Threads in speziellen Registern cached. Wir wollen aber an dieser Stelle noch nicht im Detail auf deren Funktion eingehen, sondern diese Diskussion auf die folgenden Kapitel verschieben. Es geht uns hier vielmehr darum einen Überblick über die Implementierung der Maschine als Ganzes zu vermitteln.

Die Maschine verwendet die folgenden Register:

`X`
: Das Register `X` ist ein Zeiger auf ein Feld der Länge n zur Darstellung der n festen X-Register $\mathtt{X}_0, \ldots, \mathtt{X}_{n-1}$ der Maschine. Die X-Register werden im Emulator nicht per Thread sondern einmalig für die ganze Maschine verwaltet (mehr hierzu in Kapitel 8).

`PC`
: Wie bereits eingangs erwähnt, cached der Emulator den obersten Auftrag des Threads, der gerade ausgeführt wird (mehr hierzu in Kapitel 8). Das Register `PC` repräsentiert den *Programmzähler*, der auf die Instruktion verweist, die die Maschine gerade ausführt.

`L`, `G`
: Diese Register entsprechen dem L- und G-Zeiger des gecacheten obersten Auftrags des aktuellen Threads.

`runqueue`
: Das Register verweist auf eine Schlange, die alle ausführbaren Threads (ausschließlich des gerade laufenden Threads) enthält.

`running`
: Das Register verweist auf den gerade in Ausführung befindlichen Thread.

`SP`
: Kellerzeiger, der auf den obersten Eintrag von `running` verweist.

`ap`, `mode`
: Hier handelt es sich um den Argumentzeiger `ap` und das Modus-Register `mode`, die, wie bereits in Abschnitt 4.3 beschrieben, zur optimierten Behandlung der Tupelkonstruktion eingesetzt werden.

`caseStart`, `heapSave`
: Diese beiden Register werden zur Implementierung von Case benötigt, wir gehen in den Abschnitten 3.9 und 7.6 näher auf deren Bedeutung ein.

`trail`
: Das Register verweist auf die Spur. Wie in Abschnitt 3.9 beschrieben handelt es sich dabei um einen Stapel, der temporäre Variablen-Bindungen aufnimmt,

`suspVars`
: Das Register dient zur Behandlung der Suspension von vordefinierten Funktionen und wird in Kapitel 9 beschrieben.

`heapTop`, `heapEnd`, `freelist`
: Diese Register dienen zur Verwaltung der Halde und der Freispeicherliste. Wir gehen in Kapitel 7 näher darauf ein.

6.2 Maschinencode

Der virtuelle Maschinencode wird in einem speziell dafür vorgesehenen Speicherbereich im Emulator, dem *Codebereich*, abgelegt. Für den Operationscode einer Instruktion und für jedes Argument wird je ein Wort zur Darstellung im Codebereich verwandt. Die Instruktionen besitzen also je nach Anzahl ihrer Argumente eine unterschiedliche Länge.

Eine Vereinfachung kann bei der Darstellung der unterschiedlichen Registerarten im Code vorgenommen werden. Wir erläutern das am Beispiel der Instruktion

`move` R_1 R_2

die den Inhalt des Registers R_1 in ein zweites Register R_2 kopiert. Gemäß den vier Adressierungsarten der Maschine können R_1 und R_2 dabei jeweils die Form $\texttt{X}_i$, $\texttt{L}_j$, $\texttt{G}_k$ oder $\texttt{V}_n$ annehmen. Dies bedeutet einen zusätzlichen Dekodieraufwand für jede Instruktion, die Register als Argumente verwendet, wenn man den Typ des Registers im entsprechenden Argument zusätzlich zu dessen Index kodiert. Dieser Mehraufwand läßt sich aber sparen, wenn man für jede mögliche Kombination eine eigene Instruktion definiert, also die Adressierungsarten in den Operationscode der Instruktionen kodiert. Anstelle eine generische Instruktion `move` zu verwenden, definiert man stattdessen Instruktionen `move_XX`, `move_XL`, etc. Somit wird dann aus

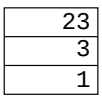
`move` $\texttt{X}_3$ $\texttt{L}_1$

die Instruktion

```
move_XL 3 1
```

Da die ursprüngliche Schreibweise allerdings deutlich lesbarer ist, wollen wir im folgenden an dieser Notation festhalten.

Wenn wir annehmen, daß die Instruktion `move_XL` den Operationscode 23 hat, dann würde die Darstellung der obigen Instruktion im Codebereich wie folgt aussehen:



Operationscode
1.Argument
2.Argument

Die Instruktion benötigt somit drei Worte zu je vier Byte. Bei genauerer Betrachtung könnte man in diesem Fall allerdings mit deutlich weniger Speicher auskommen: wenn der Instruktionssatz weniger als $2^8 = 256$ Instruktionen umfaßt, würde ein Byte zur Darstellung des Operationscodes

Tabelle 6.1 Codegröße verschiedener Anwendungen und Einsparpotential durch Halbierung Argumentgröße für Registerargumente und Opcodes.

Anwendung	Codegröße in KB	Einsparung Register	Einsparung Opcodes
Compiler	855	16,55%	15,75%
Browser	317	15,59%	15,21%
Explorer	202	16,23%	14,84%
Ozcar	146	15,33%	15,62%
Scheduler	432	15,81%	18,45%
Spedition	174	14,03%	17,50%

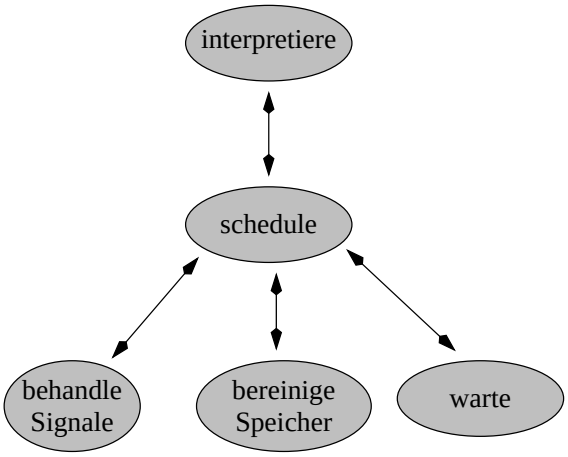
ausreichen. Mit einem einzigen Byte kommt man allerdings nicht mehr aus, wenn man die Implementierungstechnik von *threaded code* einsetzt, auf die wir in Abschnitt 6.6 eingehen werden. Wir erkaufen daher einen Laufzeitgewinn mit einem Mehrverbrauch an Speicher.

Auch für die Indizes der Registerargumente könnte man mit einem Halbwort (= 2 Byte) auskommen, wenn man den maximalen Registerindex auf $2^{16} = 65536$ beschränkt, ein Wert, der in der Praxis sicher ausreichend ist. Diese Art der Darstellung bringt aber eine Komplikation für die Darstellung der V-Register im Code mit sich: hierfür werden ja im Code direkt Zeiger in die Halde abgelegt. Bei fast allen RISC-Maschinen müssen Zeiger aber an Wortgrenzen ausgerichtet sein; auch bei modernen CISC-Prozessoren ist dies zwar nicht zwingend vorgeschrieben, wird aber zugunsten eines beschleunigten Zugriffs empfohlen. Die Einhaltung dieser Bedingung kann erreicht werden, indem man eine Instruktion immer an einer Wortgrenze beginnen läßt und die Art und Reihenfolge der Argumente aller Instruktionen so wählt, daß V-Register-Argumente stets an Wortgrenzen beginnen (was unter Umständen das Einsetzen von Dummy-Argumenten erfordert). Alternativ dazu kann auch der Interpreter selbst das Einhalten der Bedingung sicherstellen, indem er Adressen aus dem Codebereich zunächst in zwei Schritten zu je einem Halbwort lädt und die Adresse dann aus diesen beiden Halbworten zusammensetzt; dieses Vorgehen bedeutet dann aber einen verlangsamten Zugriff auf solche Argumente.

Um obigen Komplikationen aus dem Weg zu gehen, haben wir daher bei Mozart der einfacheren Variante den Vorzug gegeben, die für den Opcode und jedes Argument einer Instruktion jeweils ein eigenes Wort verwendet. Dennoch ist das mögliche Einsparpotential nicht zu vernachlässigen: Tabelle 6.1 verdeutlicht dies am Beispiel verschiedener Applikationen. Dabei lassen sich durch Verwendung eines Halbwortes für Registerindizes und den Opcode bei allen Anwendungen gleichmäßig etwas mehr als 30 % einsparen. Beim Compiler, der größten Anwendung, kommen so schon ca. 275 KB zusammen. Ein durchaus respektabler Wert, der allerdings bei stetig fallenden Speicherpreisen immer weniger ins Gewicht fällt. Zudem handelt es sich hier um einen konstanten Wert, der unabhängig ist von dem deutlich höheren Verbrauch an Haldenspeicher der Applikation zur Laufzeit.

Bei der Transformation von Code in die externe Darstellung, wie dies für das Sichern oder Verschicken von Funktionen nötig ist (vgl. Abschnitte 3.11 und 3.12), haben wir dann aber wieder auf ein möglichst kompaktes, Ressourcen schonendes Format zurückgegriffen.

Abbildung 6.1 Zustände des Emulators



6.3 Zustände des Emulators

Während der Ausführung von Programmen befindet sich der Emulator in verschiedenen Zuständen, die in Abbildung 6.1 dargestellt sind. Die zentrale Kontrolle wird dabei vom *Scheduler* übernommen. Dieser bestimmt den Thread T aus der Liste der ausführbaren Threads, der als nächstes bedient wird und übergibt T an den *Interpreter*. Der Interpreter übernimmt die Ausführung der Instruktionen von T. Er tut dies so lange bis entweder alle Aufträge von T bearbeitet wurden oder eine Signalbehandlung nötig wird. Wie in Abschnitt 3.6 bereits beschrieben, kann es sich hierbei um die Ereignisse Preemption, Speicherbereinigung oder Ein/Ausgabeaktivitäten handeln. In allen diesen Fällen geht die Kontrolle vom Interpreter wieder an den Scheduler zurück. Dieser ersetzt im Falle von Preemption den aktuellen Thread durch einen anderen, oder er führt eine *Speicherbereinigung* durch, bzw. aktiviert andernfalls die *Signalbehandlung*. Diese bestimmt die verfügbar gewordenen Ein/Ausgabekanäle und weckt die entsprechenden Threads, die auf einen dieser Kanäle suspendiert hatten.

Wenn kein ausführbarer Thread mehr existiert, aber noch offene Ein/Ausgabekanäle vorhanden sind, geht der Emulator in den Zustand *warte* (engl. idle) über, in dem darauf blockiert wird, daß einer dieser Kanäle verfügbar wird.

Im folgenden wollen wir nur auf die Arbeitsweise des Interpreters näher eingehen, da die Funktionsweise der anderen Komponenten offensichtlich ist.

6.4 C++ als Implementierungssprache

Der Emulator von Mozart wurde in C++ implementiert. Zum einen bietet C++ als objekt-orientierte Sprache bessere Abstraktionsmöglichkeiten und besseres Typchecking als C. Auf der anderen Seite unterstützt C++ aber auch die hardwarenahen Operationen, die für die effiziente Implementierung einer virtuellen Maschine unabdingbar sind. Bei sorgfältiger Verwendung von C++ müssen sich keine Effizienz-Nachteile etwa gegenüber C ergeben, wenn man an den performanzkritischen Stellen Komforteinbussen in Kauf nimmt und dort nur die reine C-Teilmenge

Abbildung 6.2 Skizze des Interpreters des Emulators

```
void interpret()
{
  loop:
    opcode = *PC
    switch (opcode) {
      case move_XL:   /* move X_i L_k */
         i = *(PC+1)
         k = *(PC+2)
         L[k] = X[i];
         PC = PC+3;
         goto loop;

      case return:
         if (runnning→isEmtpy()) {
            return;
         } else {
            <PC,L,G> = running→pop();
            goto loop;
         }

      ... /* weitere Instruktionen */

      default:
        error("opcode out of range");
     }
}
```

von C++ verwendet.

Die Wahl von C++ erwies sich auch hinsichtlich der Portabilität als richtige Entscheidung: auf praktisch allen Plattformen stehen heute gute C++ Compiler zur Verfügung; der GNU C++ Compiler bietet hier sogar eine gute kostenlose Alternative zu kommerziellen Systemen.

Bei der Darstellung von Algorithmen und Datenstrukturen werden wir daher im folgenden auf eine an C++ angelehnte Notation zurückgreifen. Wir wollen allerdings keine tiefergehenden Kenntnisse von C++ voraussetzen, sondern werden bei der Verwendung weniger offensichtlicher Konstrukte erläuternd auf diese eingehen.

6.5 Der Interpreter

Abbildung 6.2 verdeutlicht die Arbeitsweise des Interpreters, der die Ausführung der Maschineninstruktionen übernimmt. Zunächst wird der Programmzähler `PC` dereferenziert (der `*`-Operator dient in C++ zur Dereferenzierung von Zeigern) und dadurch der Operationscode der Instruktion beschafft. Dann wird in einer großen `switch`-Anweisung zu dem zugehörigen Programmstück verzweigt, das sozusagen den „Microcode“ der Instruktion darstellt. Exemplarisch haben wir hier

die Implementierung zweier einfacher Maschineninstruktionen dargestellt.

Die Instruktion `move` $\texttt{X}_i$ $\texttt{L}_k$ wird, wie in Abschnitt 6.2 bereits dargelegt, so dargestellt, daß ein Wort für den Operationscode `move_XL` und je ein Wort für die Argumente verwandt wird. Entsprechend muß der Interpreter jedes Argument aus dem Codebereich separat laden und kann dann erst das eigentliche Kopieren der Registerinhalte durchführen. Anschließend wird der Programmzähler um die Länge der Instruktion (in diesem Falle 3) erhöht. Abschließend wird durch einen Sprung zur Marke `loop` die nächste Instruktion zur Ausführung gebracht.

Abbildung 6.2 gibt anhand der Instruktion `return` ein Beispiel, wie der Interpreter auch wieder verlassen werden kann; das passiert dann, wenn der in Ausführung befindliche Thread `running` leer ist. In diesem Fall geht die Kontrolle zurück an den Scheduler. Andernfalls, wird der oberste Auftrag vom Thread genommen und die Register `PC`, `L` und `G` damit geladen. Anschließend wird dieser Auftrag wieder durch einen Sprung zur Marke `loop` zur Ausführung gebracht.

6.6 Threaded Code

Bei *threaded code* handelt es sich um eine allgemeine Technik, die zur Beschleunigung von Emulatoren eingesetzt wird [Bel73, Dew75, Kli81]. [1] Wir wollen im folgenden daher diese Technik etwas genauer vorstellen.

Bei threaded code geht es darum, den Instruktionsdispatch zu optimieren, also die Kosten, die benötigten werden, um am Ende einer Instruktion zur nächsten zu springen. Dazu betrachten wir noch einmal die entsprechende Stelle im Interpreter aus Abbildung 6.2:

```
loop:
  opcode = *PC;
  switch(opcode) {
    case move_XL:
      ...
      PC = PC+3;
      goto loop;
    ...
    default:
      error("opcode out of range");
  }
```

Eine große **`switch`** Anweisung übersetzen C++ Compiler in der Regel unter Verwendung eines Feldes `jumpTable`. Dieses enthält zu jedem gegebenen Operationscode die Adresse des zugehörigen Maschinencodes, so daß der C++ Compiler in etwa folgenden Maschinencode erzeugt, den wir hier statt in Assembler in einer C-artigen Schreibweise darstellen:

```
loop:
  opcode = *PC;
  if (opcode > LASTOPCODE) { error("opcode out of range"); }
```

[1] Obwohl diese Technik schon früh beschrieben wurde, scheint sie dennoch nicht sehr bekannt zu sein. So wird auch behauptet, daß die Implementierung des JDK 1.0 (= Java Development Kit) noch keinen threaded code verwandte, und daß erst der Wechsel eines Mitarbeiters von Quintus (einem führenden Hersteller von Prolog-Systemen) zu Sun Microsystems zur Integration dieser Technik ab JDK 1.1 führte.

```
adr = jumpTable[opcode];
goto adr;
...

move_XL:
  ...
  PC = PC+3;
  goto loop;
...
```

Der Test gleich zu Anfang ist nötig, um sicherzustellen, daß `opcode` auch im zulässigen Wertebereich liegt (wir nehmen an, daß die Konstante `LASTOPCODE` den größt-möglichen Opcode beschreibt).

Die Idee bei der Verwendung von *threaded code* besteht nun darin, im Codebereich nicht die symbolischen Operationscodes abzulegen, sondern dort direkt die absoluten Adressen der (realen) Maschineninstruktionen innerhalb des Emulators zu speichern. Der Sprung zur nächsten Instruktion reduziert sich dann auf:

```
move_XL:
  ...
  PC = PC+3;
  adr = *PC;
  goto adr;
```

Leider ist dies nun nicht mehr in C++ ausdrückbar, da C++ Marken nicht als Datenstrukturen erster Klasse zur Verfügung stellt. Der GNU C++ Compiler wurde allerdings um diese spezielle Eigenschaft erweitert, gerade um die effiziente Implementierung von Emulatoren zu erleichtern. Da C++ es erlaubt, bestimmte Quelltextteile bedingt zu übersetzen, läßt sich so derselbe Quellcode verwenden, der je nach verwandtem Compiler threaded code benutzt oder nicht. Prinzipiell ist es auch möglich, threaded code zusammen mit C++ Compilern zu verwenden, die Marken nicht als Datenstrukturen erster Klasse unterstützen. Hier muß man den Quellcode jedoch mit (wenigen) Assemblerinstruktionen anreichern, was somit Kenntnisse der zugrunde liegenden Hardwareplattform voraussetzt [CD95].

Die Verwendung von threaded code bedeutet eine deutliche Verringerung der Kosten des Instruktionsdispatches: so reduziert sich dieser im konkreten Fall beispielsweise einer SPARC basierten Maschine von 11 auf 3 Instruktionen. Da auf dieser RISC Maschine alle diese Instruktionen auch nur einen Zyklus benötigen [SPA92], ist dies mit einer entsprechenden Beschleunigung gleichzusetzen.

Ein Nachteil dieses Verfahrens besteht darin, daß sich dadurch der Speicherbedarf des Maschinencodes vergrößert: da im Code nun eine Adresse gespeichert werden muß, benötigt der Operationscode selbst nun bereits ein ganzes Wort (also 4 Byte), während zuvor ein Byte genügte (falls die virtuelle Maschine weniger als 256 Instruktionen hat). Abhilfe kann hier ein separates Feld `disptable` schaffen, das die symbolischen Operationscodes auf ihre absoluten Adressen abbildet:

```
move_XL:
  ...
  PC = PC+3;
```

Tabelle 6.2 Geschwindigkeitsteigerung verschiedener Anwendungen durch den Einsatz von threaded code.

Anwendung	Speedup
Fibonacci	1,43
Quicksort	1,41
Naive Reverse	1,19
Towers of Hanoi	1,39
Sieve	1,48
Mittel	1,38

Anwendung	Speedup
Gump Scanner	1,21
Gump Parser	1,28
Compile Modules	1,19
Browser	1,11
Explorer	1,05
Scheduler (upper)	1,02
Scheduler (prove)	0,99
Mittel	1,12

```
adr = disptable[*PC];
goto adr;
```

Diese Variante spart zwar Speicher, benötigt allerdings einen zusätzlichen Speicherzugriff.

Eine Kompromißlösung (die z.B. bei Quintus Prolog Verwendung findet) läßt sich realisieren, wenn der reale Maschinencode der Interpreterroutine `interpret` selbst weniger als 64 kB benötigt. Dann speichert man den Opcode in einem Halbwort (= 2 Byte), das den Offset zur Anfangsadresse der Prozedur `interpret` enthält:

```
move_XL:
  ...
  PC = PC+3;
  adr = interpret + *PC;
  goto adr;
```

In dieser Variante muß dann lediglich noch die Adresse `interpret` zusätzlich zur ersten Variante in ein Register geladen werden.

Tabelle 6.2 veranschaulicht den Gewinn dieser Optimierung am Beispiel verschiedener kleiner Benchmarkprogramme und anhand mehrerer großer Applikationen. Die Zeiten sind hier in Millisekunden angegeben, jeweils ohne die Zeiten für Speicherbereinigung. Gerade bei den kleinen Programmen fällt der Gewinn besonders deutlich auf; hier kann mit dieser einzigen zudem völlig sprachunabhängigen Optimierung eine Geschwindigkeitssteigerung um bis zu 48% erreicht werden.

Erwartungsgemäß fällt der Unterschied bei den größeren Anwendungen geringer aus: hier wird ein bestimmter Anteil der Laufzeit innerhalb des Laufzeitsystems (z.B. Builtins, Speicherbereinigung), der Grafik-Schnittstelle, etc. verbraucht, die nicht von threaded Code profitieren. Gerade die Constraint-Applikationen verbringen fast die gesammte Zeit mit dem Aufbau des Suchbaums und innerhalb von Propagierern, so daß sich hier praktisch kein Vorteil zeigt. Dennoch fällt der Gewinn mit zum Teil über 20% bei einzelnen Applikationen recht beachtlich aus.

Tabelle 6.3 Sicstus Prolog 3.0#3: nativer und emulierter Kode im Vergleich beim Übersetzen von Prelude Modulen von DFKI Oz 2.0 (Zeiten in Millisekunden).

Anwendung	emuliert	nativ	GC	Speedup ohne GC	Speedup incl. GC
Modules	20980	11610	5860	1,80	1,53
Browser	21200	11850	3880	1,78	1,59
Explorer	12970	7180	1610	1,80	1,65

6.7 Native Codeerzeugung

Wenn man die Implementierung des Maschinenmodells in Form eines Emulators angeht, sollte man sich bereits im Vorfeld darüber Klarheit verschaffen, ob der emulatorbasierte Ansatz prinzipiell der Richtige ist oder ob eine direkte Erzeugung von nativem Maschinencode nicht viel sinnvoller ist. Wir wollen Vor- und Nachteile der beiden Ansätze im folgenden diskutieren.

Der Vorteil bei nativem Code liegt in der Tatsache, daß die meisten virtuellen Maschinenregister direkt auf reale Register der Hardware abgebildet werden können. Insbesondere muß somit auch der Programmzähler und das damit verbundene explizite Laden der Argumente nicht simuliert werden. Haygood [Hay94] beispielsweise zeigt, daß sich beim Übergang auf nativen Code für Sicstus Prolog leicht ein Faktor von 3 gewinnen läßt. Unsere Messungen in Abschnitt 10.7 zeigen, daß dieser Faktor bei bestimmten Anwendungen in anderen Sprachen gar noch deutlich höher liegen kann.

Diese Zahlen sind allerdings sehr optimistisch, da sie sich auf sehr einfache Programme beziehen, die nur wenig mit realen Anwendungen korrelieren. So muß man in der Praxis mit deutlich geringeren Steigerungen rechnen: Tabelle 6.3 zeigt den Vergleich von emuliertem und nativem Code für eine reale Anwendung, nämlich den Compiler für DFKI Oz, der unter Sicstus Prolog 3.0 läuft. Die Tabelle zeigt jeweils die Laufzeiten (ohne Speicherbereinigungszeiten) für emulierten und nativen Code in Millisekunden. Die Spalte *GC* zeigt die Zeit für die Speicherbereinigung, die in beiden Varianten gleich hoch ist. Selbst wenn man diese Zeiten nicht einrechnet, zeigt sich, daß bei dieser speziellen Applikation höchstens mit einer Geschwindigkeitssteigerung um den Faktor 1.8 zu rechnen ist. Kalkuliert man dagegen noch die Zeiten für GC mit ein, so sinkt der Gewinn auf einen Faktor um 1.6.

Dieses Verhalten für eine spezielle Anwendung ist zwar nicht repräsentativ, allerdings kommen auch sowohl Henderson et al. [TCS95, HCS95] mit einer Steigerung von 1,42 als auch Haygood [Hay94] mit 1,56 im Mittel zu noch ungünstigeren Ergebnissen für Sicstus Prolog bei realen Anwendungen. Dies kann allerdings auch daran liegen, daß der Sicstus Compiler sowohl emulierten als auch nativen Code erzeugen kann, und der virtuelle Maschinenbefehlssatz so beiden Welten gerecht werden muß. Bessere Ergebnisse erzielt man sicher, wenn man Compiler und Befehlssatz kompromißlos auf nativen Code zuschneidet. Allerdings berichtet auch Peter Van Roy im persönlichen Gespräch, daß selbst der für seinen hocheffizienten Code bekannte Aquarius Compiler [RD92] nur unwesentlich schneller läuft, wenn er mit sich selbst übersetzt wurde (im Vergleich zu dem auf Quintus Prolog basierenden Compiler). Hierfür kann es verschiedene Gründe geben. Eine Ursache kann sein, daß Anwendungen viel Zeit mit der Ausführung von Builtins (z.B. Arithmetik) verbringen (vergleiche hierzu auch Tabelle 10.14 auf Seite 207), die in der Regel in C implementiert und somit unabhängig von der Art der Codeerzeugung sind. Es besteht auch

die Möglichkeit, daß ein erheblicher Anteil in anderen Bereichen des Laufzeitsystems verbracht wird. So kann etwa speziell bei Prolog signifikante Zeit beim Anlegen von Wahlpunkten und beim Rücksetzen verbraucht werden. Nicht zu vernachlässigen sind auch die unter Umständen hohen Kosten für die Speicherbereinigung.

Eine Beispielrechnung soll den Einfluß des Laufzeitsystems auf die Gesamtperformanz verdeutlichen. Nehmen wir an, daß bei einer Programmiersprache durch den Einsatz von nativem Code eine Geschwindigkeitssteigerung um den (hohen) Faktor 10 erzielt werden kann. Nehmen wir weiter an, daß bei einer realen Applikation ein Drittel der Zeit im Laufzeitsystem (Builtins, GC, etc.) verbracht wird (ein nicht unrealistischer Wert, wie aus Tabelle 6.3 hervorgeht). Damit werden also nur zwei Drittel der Laufzeit durch den Einsatz von nativem Code um den Faktor 10 beschleunigt. Somit ergibt sich eine effektive Beschleunigung der Applikation auf

$$\frac{1}{3} + \frac{2}{3} : 10 = \frac{2}{5}$$

Das heißt, die Applikation mit nativem Code benötigt $2/5$ der Laufzeit des emulierten Codes. Die Anwendung läuft in der Praxis also gerade mal um den Faktor 2,5 schneller.

Dem Gewinn an Laufzeit steht ein deutlich höherer Aufwand bei der Implementierung und Wartung von nativem Code gegenüber, gerade wenn eine Sprache noch so stark in der Entwicklung befindlich ist wie Oz. Insbesondere die Fehlersuche und -beseitigung gestaltet sich hier enorm schwierig. Nichtzuletzt bedarf es hier auch recht aufwendiger Einarbeitungszeiten. Demgegenüber gestaltet sich der plattformabhängige Teil bei einem Emulator minimal. Wenn überhaupt nötig, dann genügen hier einige wenige Zeilen C++-Code, die auf die unterschiedlichen Eigenschaften des Betriebssystems und der Hardware abheben. Durch die Verwendung eines Emulators gestaltete sich die Portierung von Oz auf neue Plattformen denkbar einfach: so benötigt man in der Regel oft nur ein paar Stunden, um ein voll funktionsfähiges System auf einer neuen UNIX-Plattform zu erhalten.

Beim nativen Ansatz muß man auch in der Regel mit einem deutlich erhöhten Speicherverbrauch für den Programmcode rechnen. Für Sicstus Prolog nennen Henderson et al. [TCS95] hier einen Faktor 1,7, Haygood [Hay94] kommt auf 1,7 bis 2,0. Um dies zu reduzieren, lagert man größere Teile oder ganze virtuelle Maschineninstruktionen oft in Bibliotheken aus und bringt diese durch einen Unterprogrammsprung zur Ausführung, was dann allerdings wieder Einbußen bei der Laufzeit mit sich bringt.

Für Oz ist auch schwer abschätzbar, ob der Übergang von emuliertem auf nativen Code eine gleich große Beschleunigung bringen kann, wie dies etwa bei statisch getypten funktionalen Sprachen der Fall ist. Während beim Emulator für Oz die zusätzlich benötigten Dereferenzierungsschritte und Typtests sich nicht stark bemerkbar machen, weil sie durch den allgemeinen Interpreter bedingten Overhead verwischt werden. Bei der Erzeugung von nativem Code für Oz muß daher auch sicher das Design und die Feinkörnigkeit der Maschineninstruktionen überdacht werden Am Beispiel der Fibonacci-Funktion wird dies deutlich:

```
fun fib(n) =
  case n<2 of
    true  => 1
  | false => fib(n-1) + fib(n-2)
```

die Variable `n` wird hier im Rumpf gleich drei mal in einer arithmetischen Operation verwandt. Ein Emulator wird hier jedesmal aufs neue `n` dereferenzieren, auf Integer testen und durch De-

maskieren den Wert bestimmen. Bei der nativen Codeerzeugung wird man dagegen den Instruktionssatz so feinkörnig wählen, so daß er zum Beispiel explizite Instruktionen für das Demaskieren und Dereferenzieren enthält. Für die beiden rekursiven Aufrufe könnte man im Beispiel beim nativen Ansatz auch direkt spezielle Einstiegspunkte der Funktion anspringen, indem man diese Tests überspringt. Beim Emulator ist dies allerdings viel zu teuer, so daß man hier diese Operationen gleich in die einzelnen Maschineninstruktionen integriert. Auch eine Einführung von speziellen Instruktionen, die beispielsweise ein bereits dereferenziertes Argument erwarten, scheint wenig sinnvoll, da damit der Emulator selbst deutlich vergrößert und somit wesentlich schwerer wartbar wird, bei vergleichbar nur geringem Geschwindigkeitsgewinn

Entscheidet man sich nun aber dafür zwei verschiedene Maschinenarchitekturen für nativen und emulierten Code zu verwenden, dann bringt dies auch Nachteile mit sich: es ist nicht mehr einfach möglich, in einem System sowohl nativ übersetzte als auch emulierte Funktionen zu mischen und dieses sich wechselseitig aufrufen zu lassen. Auch unter dem Aspekt der verteilten Programmierung (vgl. Abschnitt 3.12) ist die Verschickung von virtuellem Maschinencode, der zur nativen Ausführung bestimmt ist, problematisch.

Schließlich würde eine ganze Klasse von Anwendungen, bei denen es besonders stark auf hohe Performanz ankommt, von nativem Code wenig oder gar nicht profitieren. Gemeint sind Anwendungen aus dem Bereich des Constraintlösens [Sch99, Wür98]: hier wird das Gros der Rechenzeit innerhalb von Propagierern und der Suche verbracht. Diese Teile sind allerdings ohnehin in C++ implementiert und somit unabhängig vom erzeugten virtuellen Maschinencode.

Ein letzter Punkt darf beim Vergleich zwischen nativem und emulierten Code nicht vergessen werden: vielfach hat der *Speicherverbrauch* einen wichtigen Einfluß auf das Verhalten einer Applikation. Dieser ist allerdings völlig unabhängig von der verwandten Implementierungstechnik.

6.8 Verschiedene Optimierungs-Techniken für Emulatoren

Im folgenden wollen wir kurz eine Reihe weiterer Optimierungstechniken vorstellen, die speziell bei der Implementierung von Emulatoren angewandt werden können.

6.8.1 Zusammenfassung von Instruktionen

Betrachtet man den vom Compiler erzeugten Maschinencode, so stellt man fest, daß bestimmte Muster von aufeinanderfolgenden Instruktionen häufig auftreten. Beispielsweise treten oft mehrere `move` Instruktionen hintereinander auf, zum Beispiel wenn die Argumente einer Funktion in ihre L-Register gerettet oder wenn die Argumente eines Funktionsaufrufes vorbereitet werden. Dabei expandiert diese virtuelle Maschineninstruktion beispielsweise zu 7 SPARC Maschineninstruktionen (bei Verwendung von threaded code), wovon allein 3 Instruktionen für den Instruktionsdispatch benötigt werden. Faßt man nun zum Beispiel zwei aufeinanderfolgende Instruktionen

$$\texttt{move}\ R_{\texttt{i}}\ R_{\texttt{j}}$$
$$\texttt{move}\ R_{\texttt{k}}\ R_{\texttt{l}}$$

zu einer neuen Instruktion

$$\texttt{movemove}\ R_{\texttt{i}}\ R_{\texttt{j}}\ R_{\texttt{k}}\ R_{\texttt{l}}$$

zusammen, so reduzieren sich in diesem konkreten Fall die Kosten von 14 auf 11 SPARC Instruktionen.

6.8.2 Spezialisierung von Instruktionen

Bei einem Instruktionssatz, der auf einen Emulator zugeschnitten ist, wird man versuchen, die einzelnen Instruktionen möglichst grobkörnig zu wählen, um die Mehrkosten des Emulierens zu minimieren. Das hat allerdings den Nachteil, daß zum Beispiel aufeinanderfolgende Instruktionen dieselbe Variable jedesmal aufs neue dereferenzieren, deren Typ testen, etc. Durch Einführung spezieller Instruktionen, die zum Beispiel davon ausgehen, daß Ihre Argumente bereits dereferenziert sind, läßt sich dieser Aufwand verringern. Allerdings steigt dadurch unausweichlich die Größe und Komplexität des Emulators, weswegen darauf bei der Implementierung von Mozart verzichtet wurde.

6.8.3 Register-Zugriffe

Eine der häufigsten Operationen der virtuellen Maschine ist der Zugriff auf eine Variable über eines der Register. Dabei handelt es sich immer um den Zugriff auf ein Feld `A` an der Position n in der Form `A[`n`]`. Wenn jedes Feldelement die Größe k hat und A_b die Basisadresse des Feldes `A` bezeichnet, dann berechnet sich die Adresse des Elementes n zu $A_b + n \times k$. Im Fall von Registern mit der Größe eines Wortes ist hier $k = 4$, so daß bei jedem Registerzugriff der Index n zunächst mit 4 multipliziert werden muß (was der C++ Compiler in einen äquivalenten aber effizienteren Linksshift um 2 übersetzt). Günstiger ist es dann, beim Laden des Codes bereits im Codebereich statt des Indexes n gleich $n \times k$ abzulegen, so daß diese Operation nicht ständig zur Laufzeit ausgeführt werden muß, was in der Regel eine Shift-Instruktion einspart.

6.8.4 Zuordnung virtueller Maschinenregister zu realen

Die virtuelle Maschine umfaßt eine große Zahl von virtuellen Maschinenregistern, die in der Regel die Anzahl der auf einer konkreten Hardwareplattform zur Verfügung stehenden Register überschreitet, so daß es nicht möglich ist, alle virtuellen auf reale Register abzubilden. Dies ist allerdings auch gar nicht nötig, da eine Reihe von Registern (die zum Beispiel für die Threadverwaltung verwandt werden) nur selten benutzt wird. Wichtig ist also, daß man erreicht, daß eine häufig verwandte Teilmenge der virtuellen Register auf reale abgebildet werden. C++ bietet hier die Möglichkeit eine Variable als `register` zu klassifizieren, was einen Hinweis an den Compiler darstellt diese Variablen in Registern zu halten. Leider wird dieser Hinweis beispielsweise vom GNU Compiler offensichtlich ignoriert, so daß hier in der Regel eine suboptimale Verteilung vorgenommen wird. Abhelfen kann man hier indem man eine spezielle Erweiterung des GNU C++ nutzt, die es erlaubt Variablen auf reale Hardwareregister abzubilden, was allerdings wieder Kenntnisse des zugrunde liegenden Prozessors erfordert.

6.8.5 Optimierung des C++ Codes

Neben dem Emulator selbst wird auch das komplette Laufzeitsystem und somit alle vordefinierten Funktionen sinnvollerweise nicht in Maschinencode sondern in C++ implementiert. Wichtig ist hier dann eine sorgfältige Implementierung. Wegen der größeren Abstraktionsmöglichkeiten

von C++ gegenüber C ist es hier allerdings leichter suboptimalen Programmcode zu schreiben. An performanzkritischen Stellen verbieten sich so beispielsweise die Verwendung von später Bindung bei den Methoden einer Klasse oder die Übergabe von Strukturen an Funktionen mittels Wertübergabe statt Referenzübergabe.

Aber auch im Detail läßt sich durch Umschreiben des Codes oft noch viel verbessern. Hier hat es sich als einzig probates Mittel erwiesen, den vom C++ Compiler erzeugten Maschinencode zu untersuchen, um so mögliche Schwachstellen aufzuspüren.

Wir wollen dies an einem kleinen Beispiel verdeutlichen. In folgendem Codefragment, wird im Rumpf der Prozedur `p` die globale Variable `global` mehrfach verändert:

```
int p(...)
{
  ...
  global = global+n;
  int i = f(u,v);
  global = global+i;
  ...
}
```

Der C++ Compiler muß dafür sorgen, daß der Wert der Variablen immer aus dem Speicher geladen und nach jedem Ändern auch wieder direkt zurückgeschrieben wird. Wenn man als Programmierer nun weiß, daß während des Aufrufs von `f`, die Variable `global` nicht verwandt wird, dann kann man stattdessen folgenden Code verwenden:

```
int p(...)
{
  ...
  int aux = global+n;
  int i = f(u,v);
  aux = aux+i;
  ...
  global = aux;
}
```

Durch die Einführung der Hilfsvariable `aux` kann nun das mehrfache Laden und Speichern in den Hauptspeicher vermieden werden, die Hilfsvariable kann stattdessen im Keller gespeichert werden. Auf einer SPARC-Maschine kann sie wegen der dort vorhandenen Registerfenster in der Regel sogar völlig in einem Register gehalten werden. Das Beispiel zeigt aber auch, daß es oft nicht möglich ist, allen Plattformen in gleichem Maße Rechnung zu tragen: auf Intel Prozessoren ist die Anzahl der zur Verfügung stehenden Register sehr beschränkt (in der Regel höchstens 4). Dadurch kann ein intensiver Einsatz von lokalen Variablen zum cachen von globalen Werten sich ins Gegenteil verkehren, indem dann dort mehr Operationen zusätzlich über den Keller gehen müssen.

Obgleich jede der zuvor beschriebenen Maßnahmen für sich betrachtet jeweils nur kleinste Verbesserungen bringt, läßt sich in der Summe für viele Instruktionen eine deutliche Geschwindigkeitssteigerung erzielen. Betrachtet man beispielsweise den sehr oft auftretenden Fall zweier

aufeinander folgender `move` Instruktionen, so benötigen diese im unoptimierten Fall beispielsweise auf einer SPARC Maschine genau $2 \times 17 = 34$ Instruktionen. Durch Zusammenfassung in eine Instruktion lassen sich 11 Instruktionen für den Dispatch einsparen, weitere 8 werden durch threaded code eliminiert und noch einmal 4 Instruktionen durch optimierte Registerzugriffe. So bleiben in diesem konkreten Fall, bei dem sich obige Techniken besonders stark auswirken, von den anfangs 34 nur noch 11 Instruktionen übrig.

6.8.6 Cacheverhalten

Wir wollen mit einem Beispiel aus der Praxis schließen, das verdeutlicht, mit welchen Unwägbarkeiten man sich mitunter befassen muß.

Nach einer größeren Umstrukturierungsmaßnahme im Emulator mußten wir feststellen, daß bestimmte Benchmarks (z.B. Fibonacci) plötzlich um bis zu einem Faktor 1,5 langsamer liefen, obgleich die vorgenommenen Änderungen die Performanz in keiner Weise beeinflussen sollten. Eine genauere Analyse zeigte, daß sich dieses ungünstige Verhalten nur auf Intel Pentium Prozessoren der ersten Generation zeigte. Auf den neueren Pentium Pro und Pentium II Maschinen dagegen waren alter und neuer Emulator wie erwartet gleich schnell. Weiteres Experimentieren zeigte, daß die Instruktionscaches der älteren Pentium Prozessoren offensichtlich wesentlich sensibler reagierten, als die der neueren Generation. Wenn man also in der Interpreterroutine `interpret` des Emulators die Implementierung solcher virtueller Instruktionen zusammengruppiert, die in Codesequenzen auch häufig hintereinander vorkommen (so folgt auf `move` häufig `apply`), dann kann man die Wahrscheinlichkeit eines Treffers im Instruktionscache des Prozessors erhöhen. Umgekehrt kann gerade bei einfacher und kleiner Cachehardware eine ungünstige Lage häufig aufeinander folgender Instruktionen zu einer wechselseitigen Verdrängung im Cache führen.

Aber nicht nur die Lage des Maschinencodes innerhalb eines Moduls hat einen Einfluß auf das Cacheverhalten, wie ein Experiment bestätigte: Beim Fibonacci Benchmark werden relative viele arithmetische Operationen verwandt. Diese wurden in Mozart in einem separaten C++ Modul implementiert, so daß es bei Fibonacci zur Laufzeit zu vielen Sprüngen zwischen dem Arithmetikmodul und dem Modul des Interpreters kam. Beim Erzeugen des Executables des Emulators übergibt man dem Linker die Liste der einzelnen Module. Dabei hat offensichtlich die Reihenfolge, in der man die einzelnen Module in der Kommandozeile übergibt, einen direkten Einfluß auf die Lage der Module im resultierenden Executable. Indem wir das Arithmetik- und das Interpretermodul an verschiedensten Positionen in der Kommandozeile übergaben, konnten wir einen Geschwindigkeitsunterschied von einem Faktor bis zu 1,5 ausmachen.

In einem optimierten System bedarf es oft eines recht großen Aufwandes, um weitere Effizienzsteigerungen zu erzielen, die sich dann zudem in der Regel in der Größenordnung von wenigen Prozentpunkten bewegen. Die Erkenntnis, daß so unwägbare und banale Einflüsse von Außen einen unter Umständen sehr viel größeren Einfluß haben können, kann da sehr ernüchternd wirken.

Zusammenfassung

- Der *Emulator* implementiert das Maschinenmodell aus Teil II.
- Der *Scheduler* übernimmt die Ablaufkontrolle.
- *Threaded code* ist eine spezielle Implementierungstechnik für Emulatoren, die den Instruktionsdispatch deutlich beschleunigt.
- *Native Codeerzeugung* bringt in der Praxis nicht immer eine deutliche Beschleunigung. Einfache Benchmarks werden zwar oft um eine Größenordnung beschleunigt; dagegen bleiben reale Applikation je nach Intensität des Zugriffs auf das Laufzeitsystem aber oft weit dahinter zurück und laufen zum Teil kaum schneller als in einem Emulator. Demgegenüber steht ein deutlich höhere Implementierungs- und Wartungsaufwand.
- Spezielle Techniken für Emulatoren helfen, den Interpreteraufwand zu reduzieren: Instruction collapsing, Spezialisierung von Instruktionen, optimierte Register-Zugriffe, Zuordnung virtueller Maschinenregister zu realen Registern der Hardware.

Kapitel 7

Der Speicher

In diesem Kapitel werden wir auf das Speichermanagement im einzelnen eingehen. Der Darstellung und Verwaltung der Daten im Speicher ist von nicht zu unterschätzender Bedeutung für die Gesamtperformanz eines Systems. So wird hierüber nicht nur der Speicherverbrauch einer Applikation beeinflußt, sondern auch das Laufzeitverhalten hängt von einer geschickten Repräsentation der Datenstrukturen ab.

Im folgenden gehen wir in Abschnitt 7.1 zunächst näher auf die Darstellung der Halde ein. Dabei zeigen wir in Abschnitt 7.1.2, wie sich eine effiziente und wirkungsvolle Freispeicherverwaltung realisieren läßt. In Abschnitt 7.2 gehen wir dann auf die Implementierungstechnik der markierten Referenzen im Detail ein, die wir am konkreten Beispiel der Implementierung von Mozart verdeutlichen. Danach folgt in Abschnitt 7.3 eine genaue Beschreibung der Darstellung der Werte. Wir gehen dabei in Abschnitt 7.4 gesondert auf die Darstellung von Variablen ein. In Abschnitt 7.5 folgt eine Betrachtung der Besonderheiten der rationalen Unifikation. Wir schließen das Kapitel ab mit einer Beschreibung, wie effizient die Lokalität einer Variable im Wächter eines Case entschieden werden kann, was für die Entscheidung von Subsumption des Wächters wichtig ist.

7.1 Speicherverwaltung

7.1.1 Die Halde

Zur Verwaltung der dynamisch allozierten Datenstrukturen verwendet die Maschine die *Halde*. Neben den Werten, die die Variablen annehmen können, wie etwa Tupel oder Funktionen, sind das auch noch eine ganze Reihe anderer Datenstrukturen, wie etwa Suspensionslisten, Threads, etc. C++ stellt zwar mit den Operatoren `new` und `delete` und den Bibliotheksfunktionen `malloc` und `free` bereits die Möglichkeit der dynamischen Allokation und Deallokation von Speicher zur Verfügung. Die Verwendung dieser Funktionen erweist sich jedoch in der Praxis als viel zu ineffizient. Das Design der Maschine verlangt nun nicht, daß die Halde aus einem einzigen zusammenhängenden Speicherbereich besteht. Beim Start der Maschine wird daher ein größerer zusammenhängender Speicherblock (in der Praxis hat sich 1 MB als günstig erwiesen) vom Betriebssystem angefordert. Das Register `heapEnd` zeigt dann auf das Ende dieses Blockes und `heapTop` auf den noch nicht vergebenen Teil, wird also bei jeder Speicheranforderung entsprechend inkrementiert. Ist der Block aufgebraucht, wird ein neuer angefordert. Sind eine festgelegte Zahl von Blöcken aufgebraucht, wird eine Speicherbereinigung (vgl. Abschnitt 7.1.3) durchge-

führt und nicht mehr benötigte Blöcke ans Betriebssystem zurückgegeben.

In C++ läßt sich für eine Klasse elegant bestimmen, wie der Speicher bei der Erzeugung eines neuen Objektes dieser Klasse alloziert wird. Man kann eine Oberklasse `Heap` definieren, die die vordefinierten Operatoren `new` und `delete` umdefiniert. Alle Klassen, deren Speicher über die Halde verwaltet werden soll, müssen nun einfach nur von `Heap` erben. Eine explizite Freigabe von Haldenspeicher mittels `delete` ist nicht vorgesehen: wir überlassen dies der Speicherbereinigung (vgl. Abschnitt 7.1.3).

Bei der Definition des `new` Operators der Klasse `Heap` achten wir darauf, daß die Halde von oben nach unten vergeben, das heißt, daß der Wert von `heapTop` bei jeder Allokation erniedrigt wird. Dies hat einen Vorteil: für manche Datenstrukturen verlangt die Hardware, daß eine Ausrichtung an Doppelwortgrenzen erfolgt; dies ist zum Beispiel für Fließkommazahlen (die wir in L ausgespart haben) der Fall. Will man also auf der Halde Platz für eine Fließkommazahl allozieren, so muß der Haldenzeiger an einer Doppelwortgrenze ausgerichtet werden, was mindestens mit einem zusätzlichen Test verbunden ist. Wenn `heapTop` immer nur dekrementiert wird, läßt sich dies viel einfacher erreichen, indem man die niederwertigsten drei Bits von `heapTop` auf 0 setzt. `heapTop` wird also zunächst um die benötigte Größe erniedrigt und dann wird der Inhalt des Registers zusätzlich noch mit einem logischen Und mit der Maske `11...1000` verknüpft. Damit ersetzt man einen teuren bedingten Sprung durch eine billige logische Verknüpfung.

7.1.2 Freispeicherverwaltung

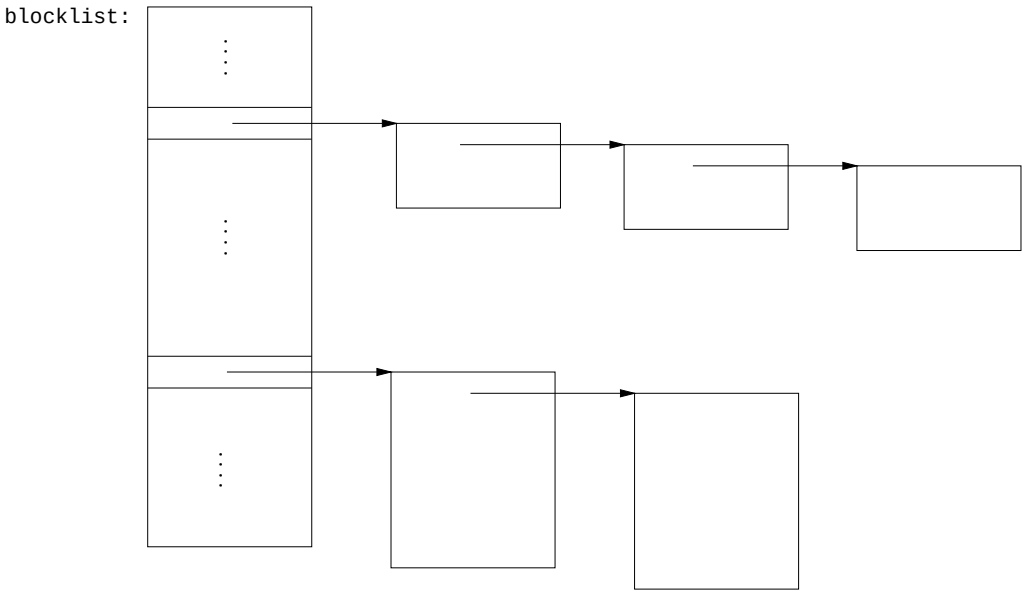
Bei den von der Halde allozierten Objekten ist im allgemeinen nicht klar, wann diese wieder freigegeben werden können. Daher wird diese Aufgabe der Speicherbereinigung überlassen. In anderen Fällen (wie zum Beispiel bei Threads (vgl. Kapitel 8) oder den L-Registersätzen) ist aber sehr wohl klar, wann die letzte Referenz auf ein Objekt verschwindet. Man könnte nun auch diese Objekte einfach der Speicherbereinigung überlassen. Man kann sie aber auch in Form einer Freispeicherliste verwalten und so die Anzahl der durchzuführenden Speicherbereinigungen unter Umständen deutlich verringern, was wiederum der Gesamtperformanz einer Applikation zugute kommt. Zudem läßt sich durch die Verwendung von Freispeicher das Cacheverhalten positiv beeinflussen, da nun bestimmte Datenstrukturen immer wieder aus dem gleichen Speichersegmenten verwandt werden. Das ist gerade für moderne Hardwarearchitekturen sehr wichtig.

Wir definieren dazu im folgenden die Klasse `FreelistMem` und werden zeigen, daß eine Freispeicherverwaltung sehr einfach realisiert werden kann und dennoch vergleichbar effizient sein kann, wie die direkte Allokation von der Halde. Wie wir weiter unten sehen werden, ist es implementierungstechnisch am einfachsten, wenn auch der Speicher der Freispeicherliste von der Halde genommen wird.

Um eine einfache und vor allem effiziente Implementierung sicherstellen zu können, werden nur Objekte bis zu einer bestimmten Größe (die durch die Konstante `maxFree` bestimmt wird) von `FreelistMem` verwaltet. Kommt eine Anforderung an `FreelistMem` zur Allokation eines Objektes, das größer ist als `maxFree`, so wird die Anforderung an die Halde weitergeleitet. Diese Beschränkung erlaubt es, ein Feld `freelist` der Länge `maxFree` zu definieren, das an der Position `freelist[`i`]` eine Liste von Blöcken mit exakt der Größe i enthält (vgl. Abbildung 7.1).

```
class FreelistMem {
  void operator delete(char *ptr, int size)
  {
```

Abbildung 7.1 Die Freispeicherliste




```
    if (size<maxFree) {
      *ptr = freelist[size];
      freelist[size] = ptr;
    }
  }
  char *operator new(int size)
  {
    if (size≥maxFree ||              /* zu gross? */
        freelist[size]==nil) {       /* Freispeicher leer */
      return Heap::new(size);        /* Allokation von der Halde */
    } else {
      char *aux = freelist[size];    /* aux ist erstes Element in der Liste */
      freelist[size] = *aux;         /* Loesche ret aus der Liste */
      return aux;
    }
  }
};
```

Eine Anforderung `FreelistMem::new(`n`)` wird dann wie folgt abgearbeitet: ist `freelist[`n`]` leer, so wird der Speicher mittels `Heap::new(`n`)` von der Halde genommen; andernfalls wird der erste Block aus `freelist[`n`]` entfernt und zurückgeliefert. Die Rückgabe von Speicher mittels `delete(`p`,`i`)` funktioniert dann analog, indem der Speicherblock, auf den p zeigt, am *Anfang* von `freelist[`i`]` eingefügt wird.

Zu Beginn sind alle Listen in `freelist` leer und werden dann mit Fortschreiten der Berechnung nach und nach gefüllt und auch wieder geleert, so daß in der Regel schon nach kurzer Zeit eine Anforderung direkt aus `freelist` und nicht mehr von der Halde befriedigt werden kann. Wäre dies nicht der Fall, so hieße dies, daß manche Objekte sehr häufig alloziert, aber nur selten

dealloziert würden; solche Objekte sollten dann besser gleich direkt von der Halde genommen werden.

Da der Speicher für die Freispeicherliste von der Halde kommt, muß darauf bei der Speicherbereinigung geachtet werden. Am einfachsten ist es, die Listen in `freelist` unmittelbar vor dem Start der Speicherbereinigung einfach zu leeren:

```
for (int i=0; i<maxFree; i++)
  freelist[i] = nil;
```

Dies bereinigt auch die Situation, daß `freelist` einmal sehr viel Speicher enthält, der nicht wiederverwandt wird.

Unter dem Gesichtspunkt der Effizienz ist die Allokation und Deallokation von der Freispeicherliste nur unwesentlich teurer als die Allokation direkt von der Halde: zum einen steht die Größe des zu allozierenden Objektes in der Regel bereits zur (C++-)Compilezeit fest, so daß die Tests `size>=maxFree` bereits zur Übersetzungszeit aufgelöst werden können. Somit reduziert sich die Deallokation auf zwei schreibende Speicherzugriffe; die Allokation kostet (falls Speicher in `freelist` vorhanden ist) einen Test und einen schreibenden und einen lesenden Speicherzugriff. Gerade bei größeren Objekten macht sich dies in der Praxis kaum bemerkbar, wenn man bedenkt, daß das Objekt ja auch noch initialisiert werden muß und über seine Lebensdauer hinweg unter Umständen viele weitere Lese- und Schreiboperationen erfolgen.

In L können die folgenden Datenstrukturen über die Freispeicherliste verwaltet werden:

L-Register Die Freigabe erfolgt genau durch die Instruktion `deallocate`.

Variablen Beim Binden einer Variablen außerhalb eines Wächters kann ihr Speicher freigegeben werden (vgl. Abschnitt 7.4).

Suspensionlisten Die Listenzellen der Suspensionslisten (nicht die Suspensionen selbst) können über die Freispeicherliste verwaltet werden. Eine Freigabe erfolgt beim Binden der Variablen und dem damit verbundenen Wecken der Suspensionen.

Threads In L kann eine Freigabe prinzipiell dann erfolgen, wenn alle Aufträge eines Threads bearbeitet wurden. Zu beachten ist, daß die Suspensionslisten von Variablen noch Referenzen auf Threads enthalten können. Wie in Abschnitt 3.7.1 beschrieben, werden diese aber beim Wecken annulliert. Ein Thread kann daher dann sicher freigegeben werden, wenn sein letzter Auftrag gerade bearbeitet wurde.

Bei der Implementierung von Mozart haben sich Freispeicherlisten als sehr nützlich erwiesen. Sie werden dort an vielen Stellen eingesetzt, wo zum Beispiel temporäre Werte alloziert werden müssen oder auch für Datenstrukturen, deren Größe sich dynamisch ändern kann, wie etwa für dynamische Arrays oder Stapel (auch von Threads), Schlangen, Tabellen etc.

Tabelle 7.1 unterstreicht die Bedeutung der Verwendung der Freispeicherliste in der Praxis. Die Tabelle zeigt den Gewinn durch der Freispeicherliste hinsichtlich des Speicherverbrauchs verschiedener Applikationen. Die erste Spalte zeigt dabei den Speicherverbrauch in Kilo-Byte auf der Halde unter Verwendung der Freispeicherliste. Die nächste Spalte zeigt den Speicherverbrauch der Applikationen ganz ohne deren Verwendung an; man erkennt, daß dadurch der Verbrauch um etwa mindestens die Hälfte bis auf mehr als das 2,7-fache ansteigen würde. Das Gros des über `freelist` verwalteten Speichers wird meist für L-Register verwandt, die die lokalen

Tabelle 7.1 Speicherverbrauch verschiedener Anwendungen (Angaben in Kilo-Byte): unter Verwendung der Freispeicherliste, ganz ohne Freispeicherverwaltung und mit Freispeicher allerdings ohne L-Register.

Anwendung	Heap	Heap ohne Freispeicher	(Zuwachs)	Heap ohne L-Register	(Zuwachs)
Laden Prelude	1.633	2.478	(51%)	1.662	(1%)
Gump Scanner	9.560	22.113	(131%)	9.930	(3%)
Gump Parser	24.357	62.903	(158%)	24.601	(1%)
Explorer	11.212	19.061	(69%)	12.353	(10%)
Browser	3.265	4.693	(43%)	3.341	(2%)
Compile Browser	29.712	73.905	(148%)	31.284	(5%)
Scheduler (upper)	13.736	39.039	(184%)	16.106	(17%)
Scheduler (prove)	14.681	24.687	(68%)	22.172	(51%)
Spedition	2.001	3.192	(59%)	2.692	(34%)

Variablen einer Funktion aufnehmen (vgl. Kapitel 3). Da diese in anderen Sprachen über den Keller alloziert werden, zeigt die letzte Spalte in Tabelle 7.1 die Einsparungen, wenn man den für L-Register benötigten Speicher herausrechnet. Hier ergibt sich nunmehr für manche Applikationen nur noch ein geringer Vorteil. Dagegen profitieren gerade diejenigen Applikationen immer noch stark, die zum Beispiel viele Threads erzeugen, aber gerade auch der Scheduler, der als Constraint-Applikation viele Constraint-Variablen erzeugt, deren Speicher nach dem Binden wieder freigegeben wird. So kann hier die Einsparung immerhin bei bis zu 51% liegen, was gerade bei den sehr laufzeitintensiven Constraint-Anwendungen besonderns wichtig ist, da dies auch gleichzeitig eine Reduzierung der Laufzeit durch weniger häufig durchgeführte Speicherbereinigungen bedeutet.

Somit kann die Verwendung der Freispeicherliste zu einer deutlichen Reduzierung des Gesamtspeicherverbrauchs einer Applikation beitragen. Andere Ansätze wie zum Beispiel beim CPS (continuation-passing style) [App92] verfolgen eine ganz andere Strategie: hier geht man davon aus, daß die Kosten für das Einsparen von Speicher höher liegen können, als die Kosten für zusätzliche Läufe der Speicherbereinigung. Dies gilt aber nur dann, wenn man geeignete Verfahren zur Speichbereinigung einsetzt, wie zum Beispiel die generationsbasierte Speicherbereinigung [LH83]. Der Einsatz solcher Verfahren für Mozart ist aber noch nicht realisiert, insbesondere ist noch nicht klar, ob dies überhaupt möglich ist und wenn ja mit welchem Aufwand und welchem Gewinn dies einhergeht (vgl. auch Abschnitt 7.1.3).

7.1.3 Speicherbereinigung

Wenn die Größe der Halde einen gewissen Schwellwert *gcMargin* (dieser und alle folgenden Parameter können vom Benutzer eingestellt werden) übersteigt, wird eine Speicherbereinigung veranlaßt. Diese wird allerdings nicht direkt innerhalb der Allokationsroutine aufgerufen, da hier im allgemeinen nicht alle Einstiegspunkte für die Speicherbereinigung klar definiert sind. Deshalb wird hier nur eine Marke gesetzt, die später an geeigneter Stelle (beim Funktionsaufruf,

vgl. Kapitel 8) abgetestet wird.[1] Wenn nach der Speicherbereinigung noch mehr (weniger) als ein bestimmter Prozentsatz der Halde belegt ist, wird *gcMargin* um einen bestimmten Faktor erhöht (erniedrigt). Bei geschickter Wahl dieser Parameter läßt sich so erreichen, daß der Anteil der Speicherbereinigung an der Laufzeit einer Applikation auf ein adäquates Maß reduziert wird: speicherhungrigen Anwendungen stellt man dabei eine große Halde zur Verfügung und erreicht so, daß diese wegen seltenerer Speicherbereinigungen schneller wieder terminieren und die belegten Ressourcen freigeben. Diesem Vorgehen liegt die Beobachtung zu Grunde, daß CPU Zeit kostbarer ist als Hauptspeicher.

Bei der Implementierung der Speicherbereinigung in Mozart kam ein einfacher Stop und Copy-Algorithmus [Bak78] zum Einsatz: alle erreichbaren Haldenzellen werden in einen neuen Speicherbereich kopiert und nach der Speicherbereinigung wird der ursprünglich von der Halde belegte Speicherbereich wieder freigegeben. Beim Kopieren kann eine wichtige Optimierung vorgenommen werden: Referenzketten können verkürzt und in der Regel (wenn sie nicht in einer Variable enden, vgl. Abschnitt 7.4) auch ganz eliminiert werden. Selbstverständlich können auch suspendierte Threads beseitigt werden, wenn sie nur noch in den Suspensionslisten von nicht mehr referierbaren Variablen vorkommen.

Wie bereits in Abschnitt 4.1.6 angesprochen, umfaßt die Speicherbereinigung in Mozart auch den Codebereich.

Fortgeschrittenere Techniken, wie etwa generationale Speicherbereinigung [LH83] wurden noch nicht implementiert: eine Implementierung gerade dieser Technik ist nicht trivial, da sehr viele unterschiedliche Datenstrukturen auf der Halde alloziert werden und prinzipiell bei jeder Änderung eines Zeigers dann zusätzlich getestet werden muß, auf welche Generation dieser nun verweist. Wie Abbildung 7.2 verdeutlicht spielt zudem der Anteil der Speicherbereinigung an der Gesamtlaufzeit vieler Applikationen nur eine untergeordnete Rolle; hier zahlt sich die oben erwähnte Philosophie aus, speicherhungrigen Anwendungen lieber eine ausreichend große Halde zu spendieren anstatt häufiger Speicherbereinigungen durchzuführen. So zeigt Tabelle 7.2, daß bei den dort gemessenen Anwendungen der Anteil für die Speicherbereinigung stets weniger als 10 Prozent an der Gesamtlaufzeit beträgt, oft liegt diese Zahl sogar noch deutlich darunter. Hier besteht also momentan kein Handlungsbedarfbedarf für die Integration fortgeschrittener Techniken.

Die Situation sähe allerdings bei der Erzeugung von nativem Code anders aus: da die Geschwindigkeit der Speicherbereinigung davon nicht beeinflußt wird, andererseits die Ausführungsgeschwindigkeit insgesamt gesteigert wird, muß dann unter Umständen mit einem deutlichen Anstieg der relativen Kosten für die Speicherbereinigung gerechnet werden. Wie stark dieser Anstieg dann aber bei realen Anwendungen wirklich ist (vgl. Abschnitt 6.7), muß dann erst noch ermittelt werden, um festzustellen, ob Bedarf für Nachbesserungen besteht.

Auch die absolute Geschwindigkeit der Speicherbereinigung ist recht hoch: so können auf einem relativ langsamen System (Pentium 100 MHz) immer noch circa 15 MB aktiven Haldenspeichers innerhalb einer CPU Sekunde kopiert werden. Das bedeutet, daß in der Praxis dem Benutzer nur sehr selten das Einsetzen einer Speicherbereinigung als deutliches Stocken der Applikation unangenehm auffällt, und er dies in der Regel überhaupt nicht wahrnimmt.

[1] Dabei wird davon ausgegangen, daß der virtuelle Speicher des Betriebssystems bis dahin ausreicht, andernfalls muß mit einer Fehlermeldung abgebrochen werden.

Tabelle 7.2 Anteil der Zeit für Speicherbereinigung an der Gesamtlaufzeit verschiedener Anwendungen.

Anwendung	Anteil GC
Compile Browser	3,7%
Gump Scanner	3,8%
Gump Parser	4,2%
Explorer	7,7%
Browser	1,5%
Scheduler (upper)	8,7%
Scheduler (prove)	6,9%
Spedition	7,0%

7.2 Markierte Referenzen

Da es sich bei L um eine dynamisch getypte Sprache handelt, muß zur Laufzeit für alle Werte deren Typ erkennbar sein. Wenn nun aber jeder Wert, ein Typfeld enthält, dann wird dadurch ein komplettes Wort belegt, obwohl dazu lediglich wenige Bits innerhalb eines Wortes benötigt werden.

Die gängige Technik, dieses Manko zu beseitigen, besteht darin, die Typmarkierungen aus den Objekten heraus in die Referenzen auf diese Objekte zu verlagern. In der Darstellung des Graphenmodells aus Kapitel 2 würde das gerade dem Vorgehen entsprechen, daß man die Kanten des Graphen zusätzlich mit den Typen der Knoten, auf die die Kanten verweisen, markiert.

Bei der Verwendung *markierter Referenzen* (tagged references), enthalten somit die X-, L- und G-Register der Maschine und die Argumente von Tupeln Worte der folgenden Form:

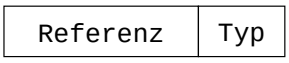
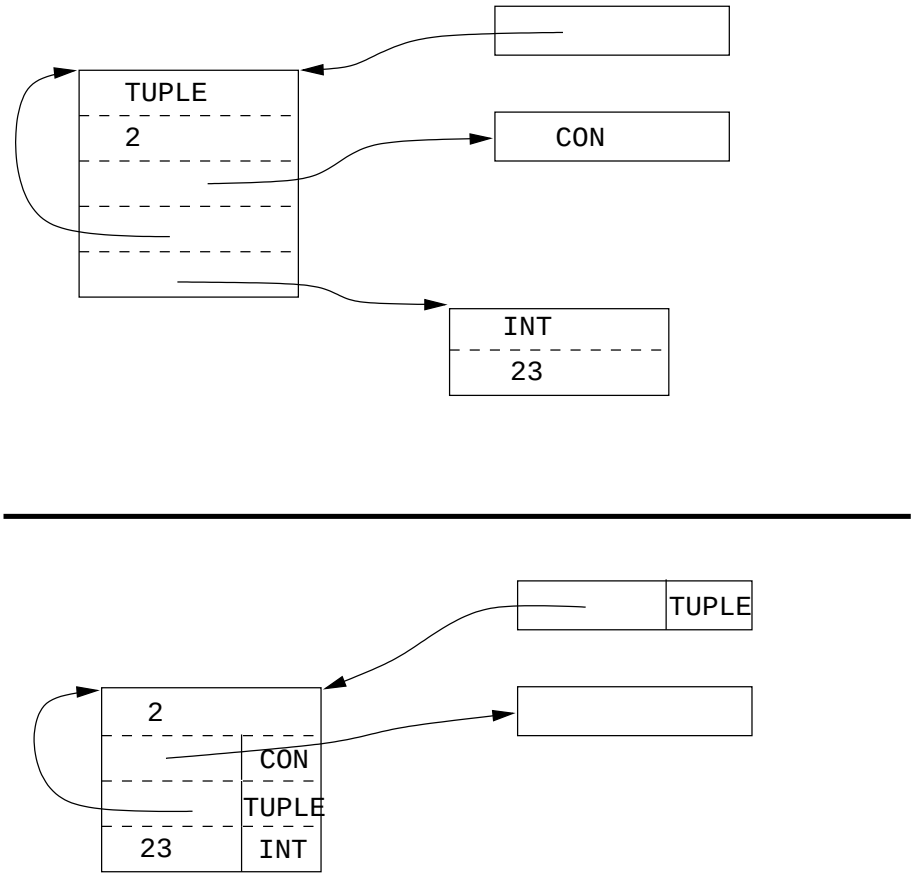


Abbildung 7.2 zeigt vergleichend die Darstellung eines Tupels einmal in der Form markierter Objekte und einmal mit markierten Zeigern. Man beachte, daß der Typ eines Speicherobjektes (z.B. eines Tupels) nun nicht mehr an diesem selbst erkennbar ist, sondern lediglich an den Verweisen darauf.

Beim Übergang auf markierte Referenzen sind zwei Punkte besonders zu beachten: dies betrifft die Darstellung von Variablen und die von ganzen Zahlen. Wie aus Abbildung 7.2 hervorgeht, muß das `Referenz`-Feld nicht immer einen Zeiger enthalten, im Falle von ganzen Zahlen enthält es direkt deren Wert. Dadurch brauchen arithmetische Operationen (Beispiel Addition) kein neues Objekt auf der Halde zu allozieren, sondern können ihre Ergebnisse direkt in Registern zurückliefern.

Zum anderen funktioniert die Verlagerung der Marken in die Zeiger nur deshalb, weil sich der Typ eines Knotens während der Laufzeit nicht mehr ändern kann. So kann etwa ein Tupel t nicht mehr in eine Zahl mutieren; andernfalls müßten alle Referenzen auf t nun aktualisiert werden, was nicht praktikabel ist. Der einzige Knotentyp, dessen Wert sich ändern kann, ist eine Variable:

Abbildung 7.2 Darstellung eines Tupels: oben mit markierten Objekten unten mit markierten Zeigern.



beim Binden wird aus einem VAR-Knoten ein REF-Knoten, was eine besondere Behandlung von Variable-Knoten bedingt.

Wir werden in den Abschnitten 7.3.1 und 7.4 noch ausführlich auf die beiden obigen Punkte eingehen.

Insgesamt ergeben sich die folgenden Vor- und Nachteile von markierten Referenzen gegenüber markierten Objekten, die wir hier nur kurz zusammenfassen wollen und im späteren noch vertiefen werden:

+ Reduzierter Speicherverbrauch, da das Typfeld in Objekten eingespart wird.
+ Weniger Speicherzugriffe, da der Typ bereits in den Registern vorliegt. Damit schnellerer Typtest: Demaskieren statt Speicherzugriff.
+ Reduzierter Speicherverbrauch bei Integern (vgl. Abschnitt 7.3.1).
+ Kürzere Referenzketten beim Binden von Variablen: Einführen von REF-Zellen kann vielfach gespart werden (siehe Abschnitt 7.4).
– Dereferenzieren eines Zeigers benötigt Demaskieren des Typfeldes.

- Verringerung des verwendbaren Adressraumes.
- Verringerung des Wertebereichs für ganze Zahlen.
- Darstellung und Binden von Variablen wird komplexer (siehe Abschnitt 7.4).

7.2.1 Wahl und Darstellung der Marken

Die Performanz vieler häufig verwendeter zentraler Operationen der Maschine, wie das Dereferenzieren, Typtesten, Maskieren und Demaskieren von Referenzen, hängen entscheidend von der Wahl der Repräsentation markierter Referenzen ab, so daß deren geschickte Implementierung einen wichtigen Beitrag zur Gesamteffizienz des Systems leistet. Gudeman [Gud93] gibt einen guten Überblick über die Techniken, die hier zur Anwendung kommen können. Wir werden in diesem Abschnitt die Implementierung solcher markierter Referenzen genauer darstellen. Dabei wird nicht wie bisher lediglich auf die für die Darstellung der vereinfachten Teilsprache L benötigten Datenstrukturen eingegangen, sondern wir wollen vielmehr anhand aller für die Implementierung von Mozart benötigten Objekte, auf die Anforderungen einer konkreten Implementierung eingehen.

Auf 32 Bit Architekturen bietet es sich an, ein Maschinenwort zur Darstellung von markierten Referenzen zu wählen (was dem C++ Datentyp `int` entspricht). Auch auf 64-bittigen Architekturen wurde in Mozart eine Darstellung von markierten Referenzen über 32 Bit breite Speicherzellen verwandt: ein Rückgriff auf 64 Bit würde zwar den zur Verfügung stehenden Adreßraum erhöhen (der wie weiter unten beschrieben in Oz bei 32 Bit bei immerhin 1 GB liegt) aber gleichzeitig eine Verdopplung des Speicherverbrauchs jeder Anwendung nach sich ziehen.

Für die Darstellung der Marke werden, wie weiter unten beschrieben, 4 Bit benötigt. Somit bleiben 28 Bit für das `Referenz`-Feld. Da moderne Prozessoren in der Regel schneller auf die unteren Bits eines Wortes zugreifen können [Gud93], wird die Marke in den niederwertigen Bits untergebracht.

Die Erzeugung einer markierten Referenz aus einem Zeiger und der Marke benötigt zwei Zyklen (ein Shift um 4 nach Links und ein logisches Oder). Dagegen benötigt das Beschaffen der Marke (logisches Und) oder des Zeigers (Shift nach Rechts) je nur einen Zyklus.

Beim Erzeugen einer markierten Referenz ist allerdings folgendes zu beachten: auf vielen Architekturen (z.B. MIPS, Power-PC, HPPA, Alpha) beginnt die Basisadresse des Datensegments eines Prozesses nicht bei der Adresse 0. Hier sind einige der höherwertigen Bits einer Adresse immer gesetzt (z.B. Bit 29 bei MIPS); zum Teil kann die Basisadresse des Datensegments auch durch Optionen des Binders beeinflußt werden. Das bedeutet, daß durch das Verschieben des Zeigers nach Links unter Umständen signifikante Bits einer Adresse eliminiert werden. Daher muß auf diesen Plattformen bei der inversen Operation auch noch die Basisadresse wieder hinzugefügt werden, was in der Regel ein oder zwei zusätzliche Zyklen kostet.

7.2.2 Das Tag-Schema von Mozart

Tabelle 7.3 zeigt das Schema für markierte Referenzen von Mozart. Mit einer Ausnahme enthalten die Wertefelder stets Zeiger auf verschiedene Objekte. Lediglich für kleine ganze Zahlen (`INT`) wird der Wert gleich im Wertefeld abgelegt, sofern 28 Bit ausreichen. Sonst wird auf eine Darstellung als Zahl mit beliebiger Stelligkeit übergegangen (vgl. Abschnitt 7.3.1).

Tabelle 7.3 Das Tag-Schema von Mozart 3.0

Marke	Bitmuster	Beschreibung
UVAR	0001	Variable ohne Suspensionsliste
CVAR	0101	Variable mit Suspensionsliste oder Constraints
INT	0110	kleine ganze Zahl
FLOAT	1011	Fließkomma Zahl
LIST	0010	Liste
REC	0011	Tupel/Record
LIT	1111	Literal
OTHER	1010	Sonstige (Zelle, Prozedur, Thread, etc.)
GCTAG	1101	Marke für Speicherbereinigung
REF	XX00	Gebundene Variable (= Referenz)

Zusammenfassung von Objekten

Obgleich in 4 Bit prinzipiell 16 verschiedene Marken dargestellt werden können, reicht dies für Mozart nicht aus. Daher wurden verschiedene Objekte unter einer speziellen Marke OTHER zusammengefaßt. Hier verweist das Wertefeld auf ein Objekt das selbst wieder eine eigene sekundäre Marke zur weiteren Unterscheidung enthält. Der Test auf Gleichheit wird dadurch nicht negativ beeinflußt, da die Gleichheit fast aller dieser Objekte über ihre Lage im Speicher definiert ist. Ein Typtest für diese Objekte muß allerdings zweistufig erfolgen und ist somit teurer als für andere Werte, so daß hier nur Objekte verwandt wurden, für die dies weniger kritisch ist. Beispielsweise werden Prozeduren so dargestellt: die mit Abstand häufigste Operation auf Prozeduren, die einen Typtest verlangt, ist die Applikation. Diese Tests können aber wie in Kapitel 4 beschrieben in der Regel durch verschiedene Optimierungstechniken eliminiert werden.

Speicherbereinigung

Um Zyklen zu erkennen und damit Objekte nicht zweimal kopiert werden, müssen bei der Speicherbereinigung Objekte die bereits abgearbeitet wurden, speziell markiert werden. Hierzu wird oft ein eigenes Bit innerhalb eines Wortes verwandt. Dies bedeutet aber, daß sich dadurch der Adreßbereich und damit die maximale Größe der Halde halbiert. Für Mozart hieße das (bei 4 Bit für die Marke und 1 Bit für die Speicherbereinigung), daß die Größe der Halde auf $2^{27} = 128$ MB beschränkt wäre. Dies ist eindeutig zu wenig für viele Anwendungen insbesondere im Bereich des Constraintlösens. Daher gingen wir in Mozart dazu über, statt eines speziellen Bits für die Speicherbereinigung lediglich eine eigene Marke GCTAG zu verwenden. Das bedeutet dann allerdings, daß der Test auf Gesetztheit der Marke geringfügig teurer wird, während das Demaskieren in der Regel keine Extrakosten verursacht.

7.2.3 Wahl der Marken

Die Wahl der Marken, das heißt die Zuordnung der verschiedenen Bitmuster zu den einzelnen Objekten wurde mit Bedacht gewählt, um die Kosten für Typtests, Dereferenzierung und Demaskierung zu minimieren. Hier wurde davon ausgegangen, daß auf modernen Prozessoren sehr billig (d.h. in einem Maschinenzyklus) festgestellt werden kann, ob eine bestimmte Menge von Bits innerhalb eines Wortes *nicht* gesetzt ist.

Listen

Wie in Abschnitt 4.3.4 ausgeführt, werden Tupel, die Listen darstellen, unter Verwendung einer eigenen Marke `LIST` gesondert repräsentiert.

Üblicherweise muß für einen Typtest zuerst die Marke extrahiert werden, bevor der eigentliche Vergleich erfolgen kann. Für die besonders häufig verwandten Listen wurde das Bitmuster `0010` gewählt. So kann in einem Schritt ohne Demaskierung getestet werden, ob es sich um eine Liste handelt; das gilt genau dann, wenn ein bitweises logisches Und mit der Maske `11...1101` den Wert 0 ergibt.

Variablen

Wir werden in Abschnitt 7.4 auf die Darstellung von Variablen genauer eingehen. Dort werden wir sehen, daß es in Mozart zwei Arten der Darstellung für Variablen gibt, die mit den Marken `UVAR` und `CVAR` versehen sind. Um nun zu Testen, ob es sich bei einer markierten Referenz um eine Variable handelt, muß nicht nacheinander auf `UVAR` und dann auf `CVAR` getestet werden. Die Bitmuster dieser Marken wurden so gewählt, daß sie die einzigen sind, deren zweites Bit nicht gesetzt ist (das gilt zwar auch für `GCTAG` und `REF`, erstere wird aber nur temporär während der Speicherbereinigung verwandt; eine `REF` Zelle kann auch nicht vorliegen, da vor einem Typtest stets dereferenziert werden muß). So kann ohne Demaskieren in einem Zyklus auf das Vorhandensein einer Variable getestet werden: analog zu Listen gilt dies genau dann, wenn ein logisches Und mit `00...0010` wieder 0 ergibt.

Da die Variablen, die am häufigsten verwandt werden, vom Typ `UVAR` sind, wurde auch deren Typtest optimiert. Ihr Bitmuster wurde analog zu dem für Listen so gewählt, daß nur ein Bit darin gesetzt ist, so daß ein logisches Und mit `11...0111` genau in diesem Fall 0 liefert.

Referenzen und Dereferenzieren

Besonderes Augenmerk wurde auf die effiziente Darstellung von gebundenen Variablen (`REF`-Zellen) und damit auch auf die Dereferenzierungs-Operation gelegt. Zunächst wurde sichergestellt, daß alle Objekte, die von der Halde alloziert werden, an Wortgrenzen ausgerichtet sind. Dies ist auf RISC Maschinen in der Regel ohnehin zwingend vorgeschrieben [KH92, SPA92, MSS91] und auch für die Intel Prozessoren wird dies aus Performanzgründen empfohlen [Int90]. Da ein Wort mindestens 32 Bit = 4 Byte groß ist, bedeutet das, daß die beiden niederwertigen Bits aller Zeiger in die Halde auf `0` gesetzt sind.

Bei der Wahl der Marken für alle nicht-`REF` Zellen wurde darauf geachtet, daß mindestens eines der beiden unteren Bits verschieden von `0` ist. Dadurch brauchte die Marke `REF` nicht mehr ex-

plizit repräsentiert zu werden: das Erzeugen einer `REF`-Zelle aus einem Haldenzeiger `p` geschieht dann ohne weitere Schiebeoperationen, indem `p` selbst direkt als markierte Referenz interpretiert wird. Eine `REF`-Zelle liegt also genau dann vor, wenn die beiden unteren Bits den Wert `0` haben (was wie oben beschrieben in einem Zyklus zu testen ist). Die häufig verwandte Operation des Dereferenzierens, die jedem lesenden Zugriff auf eine markierte Referenz vorausgehen muß, konnte somit ohne Demaskierungs- und Schiebeoperationen realisiert werden.

7.2.4 Vergrößerung des Adreßraums

Bei einer Breite der Marken von 4 Bit bleiben für das Wertefeld noch 28 Bit, womit sich die maximal adressierbare Größe der Halde zu $2^{28} = 256$ MB errechnet, ein Wert der unter heutigen Gesichtspunkten je nach Anwendung mitunter nicht ausreicht.[2]

Man kann den verfügbaren Adreßraum allerdings vergrößern, indem man sich die Tatsache zu Nutze macht, daß mit Ausnahme von ganzen Zahlen alle anderen Wertefelder Zeiger enthalten. Da diese Zeiger an Wortgrenzen ausgerichtet sind, sind deren beiden niederwertige Bits stets `0`. Diese Information braucht also nicht abgespeichert zu werden. Beim Erzeugen einer markierten Referenz aus einem Zeiger `p` braucht dieser also nicht um 4 sondern lediglich um 2 Bit nach links verschoben zu werden, danach werden die unteren beiden Nullen durch die oberen beiden Bits der Marke überschrieben.

Dies erzeugt keine Mehrkosten. Auch muß das Beschaffen der Marke nicht modifiziert werden. Lediglich das Beschaffen des Zeigers muß angepaßt werden: nach dem Schieben um 2 nach rechts müssen die unteren beiden Bits noch auf `0` gesetzt werden, was Mehrkosten von einem Maschinenzyklus verursacht. Die Marken für Listen, Records und die häufigen `UVAR`-Variablen wurden nun so gewählt, daß die beiden oberen Bits nicht gesetzt sind. Somit braucht die Annullierung der beiden niederwertigen Bits in den Zeigern für diese besonders häufig verwandten Datenstrukturen nicht vorgenommen zu werden, so daß auch hier keine Mehrkosten entstehen.

Durch diese Maßnahme vervierfacht sich der zur Verfügung stehende Adreßraum auf nun 1 GB.

Ein weiterer Vorteil ergibt sich für Plattformen, bei denen wie weiter oben beschrieben die Halde nicht bei der Adresse `0` beginnt: beim Beschaffen des `Referenz`-Teils einer markierten Referenz kann auf das Restaurieren der oberen Bits der Haldenadresse dann verzichtet werden, wenn die beiden höchstwertigen Bits in der Startadresse der Halde nicht gesetzt sind. Dies ist für alle von Mozart unterstützten Plattformen mit Ausnahme von HPPA der Fall.

7.3 Darstellung der Werte

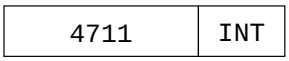
Wir wollen in diesem Abschnitt auf die Darstellung der verschiedenen Werte von L in der Maschine eingehen. Da es sich bei L um eine Teilsprache von Oz handelt, kann bei L für bestimmte Werte eine gegenüber vollem Oz vereinfachte Darstellung gewählt werden. Möglich ist dies, da L zum einen keine tiefen Wächter unterstützt: in vollem Oz müssen viele Werte noch eine Referenz auf den Berechnungsraum tragen, in dem sie erzeugt wurden: bei Variablen ist dies nötig, um Subsumption testen zu können; viele andere Werte benötigen einen solchen Zeiger, damit beim Kopieren in Zusammenhang mit der Suche entschieden werden kann, ob eine Kopie angefertigt werden muß [Sch99]. Eine weitere Vereinfachung für L besteht darin, daß L weniger Werte als

[2] SICStus Prolog verliert sogar 5 Bit [Car91], was den Adreßbereich auf 128 MB einschränkt.

Oz enthält und somit auf die Einführung von sekundären Marken über die primäre Marke OTHER (vgl. Abschnitt 7.2.1) verzichtet werden kann. Für die Werte von L werden wir daher im folgenden die konkrete Darstellung in der Implementierung von Mozart beschreiben und werden kurz auf mögliche optimierte Darstellungen im Zusammenhang mit L eingehen.

7.3.1 Zahlen

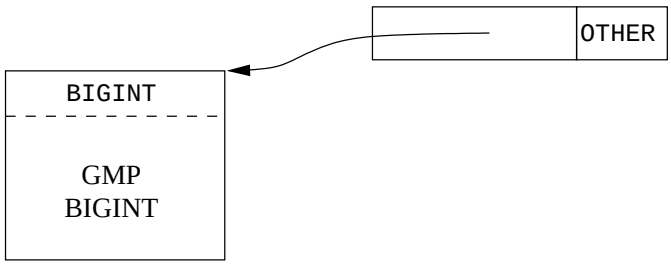
Die Darstellung von ganzen Zahlen in einer markierten Referenz erfolgt unter Verwendung der Marke INT. Der Referenzteil enthält hier als einzige Ausnahme keinen Zeiger sondern direkt den Wert der Zahl. Die Zahl 4711 wird also wie folgt dargestellt:



Für die Darstellung der Zahl im Referenzteil könnte man nun etwa ein eigenes Bit für das Vorzeichen vorsehen und im Rest den Betrag der Zahl darstellen. Wir wählen dagegen direkt die übliche Zweierkomplement-Darstellung im Referenzteil. Das hat verschiedene Vorteile: einerseits kann der Wert der Zahl sehr einfach durch einen *arithmetischen* Rechtsshift (im Gegensatz zum logischen Rechtsshift werden hier führende Einsen nachgezogen) beschafft werden. Auch die inverse Operation kommt mit einem Linksshift und der Addition der Marke INT aus. Bestimmte arithmetische Operationen wie etwa die Vergleichsoperationen kommen gar ganz ohne explizites Beschaffen der Ganzzahl-Werte zweier markierter Referenzen aus: hier kann man direkt zwei markierte Referenzen als Zahlen interpretieren und miteinander vergleichen.

Große ganze Zahlen

Dadurch, daß man Teile eines Wortes für die Marken verwenden muß, reduziert sich der bereits ohnehin endlich beschränkte darstellbare Zahlbereich noch weiter. Mozart erlaubt dem Benutzer daher die Verwendung beliebig großer ganzer Zahlen. Dazu verwendet die Implementierung zwei unterschiedliche interne Repräsentationen für ganze Zahlen, die für den Benutzer völlig transparent ist. Reicht zur Darstellung einer Zahl n der verfügbare Wertebereich nicht aus, so wird diese als große Zahl unter Verwendung der sekundären Marke BIGINT dargestellt:



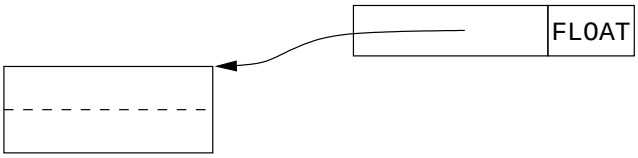
Der Referenzteil zeigt dabei auf ein Objekt, das die Zahl n darstellt. Für die Darstellung und die Implementierung der arithmetischen Operationen, haben wir in Mozart hierzu auf ein vorhandenes Paket, die GNU MP-Bibliothek [Gra93], zurückgegriffen.

Die Einführung von beliebig großen ganzen Zahlen, bedeutet nun aber, daß nach fast jeder arithmetischen Operation geprüft werden muß, welche Darstellung für das Ergebnis gewählt werden

muß, wodurch fast jede dieser Operationen verlangsamt wird, obgleich in der Praxis nur recht selten große Zahlen wirklich benötigt werden. Dieser Mehraufwand macht sich allerdings bei einem Emulator nur wenig bemerkbar.

Fließkommazahlen

Neben ganzen Zahlen erlaubt Mozart auch die Verwendung von *Fließkommazahlen*:



Der Zeigerteil verweist auf die maschineninterne Darstellung von Fließkommazahlen, die üblicherweise zwei Worte verwendet. Während bei arithmetischen Operationen auf (kleinen) ganzen Zahlen keinerlei Haldenspeicher für das Ergebnis alloziert werden muß, da diese in einem Maschinenwort samt Marke dargestellt werden können, muß dagegen für Fließkommazahlen stets ein Doppelwort alloziert werden.

Prinzipiell wäre auch hier eine Einschränkung von Fließkommazahlen auf eine Länge von 28 Bit denkbar (wie dies etwa auch bei Aquarius Prolog geschehen ist). Allerdings erwartet der Benutzer eine Implementierung der Fließkommaarithmetik, die dem IEEE Standard entspricht, so daß diese Möglichkeit ausscheidet.

Somit bedeutet die Verwendung von Fließkommazahlen in Oz auch immer eine deutliche Erhöhung des Speicherbedarfs. Statisch getypte Sprachen sind hier deutlich im Vorteil: während in Oz Zwischenergebnisse stets auf die Halde kommen und die L-Register nur Referenzen darauf aufnehmen, kann man in getypten Sprachen erkennen, daß eine Variable eine Fließkommazahl aufnimmt und entsprechend eine sogenannte „ungeboxte “ Darstellung wählen [Ler97].

Obwohl statisch getypte Sprachen bei der Verwendung von Fließkommazahlen, wie oben diskutiert, deutlich im Vorteil sein müßten, wird dies in der Praxis nicht unbedingt bestätigt: wie wir in den Abschnitten 10.7.2 und 10.7.6 sehen werden, können beispielsweise die ML Implementierungen (Ausnahme ist lediglich SML/NJ) Fließkommazahlen offensichtlich nur selten in Registern halten, sondern allokieren diese oft neu auf der Halde. So steigt auch hier der Speicherverbrauch sofort drastisch an, wenn von ganzzahliger auf Fließkomma-Arithmetik übergegangen wird.

7.3.2 Literale

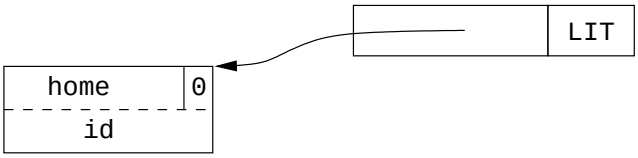
Oz bietet neben Namen (= Konstruktoren in L) auch Atome (= Zeichenketten). Beide zusammen werden als *Literale* bezeichnet.

Die Operation, die am häufigsten auf Atome und Namen angewandt wird, ist der Vergleich mit einem anderen Wert. Daher verwenden Atome und Namen unter Verwendung einer gemeinsamen Marke `LIT` dargestellt. Zur Unterscheidung in Namen und Atome wird dann noch ein Bit innerhalb der diese Werte repräsentierenden Objekte vorgesehen. Atome werden in der Atomtabelle verwaltet, die Zeichenketten auf die unten beschriebene interne Darstellung abbildet, so daß zwei gleiche Zeichenketten stets durch denselben Atomknoten dargestellt werden. Somit

können Literale einfach auf Gleichheit verglichen werden: es reicht ein Vergleich der beiden markierten Referenzen, die Inhalte der zugehörigen Strukturen müssen nicht weiter betrachtet werden. Allerdings wird dadurch die (in der Praxis seltene) dynamische Erzeugung von Atomen (wie dies in Mozart etwa durch die Konkatenation der Zeichenketten zweier Atome möglich ist), entsprechend teurer, da erst ein neuer Eintrag in der Atomtabelle erzeugt werden muß.

Namen

In Mozart werden Namen wie folgt dargestellt:

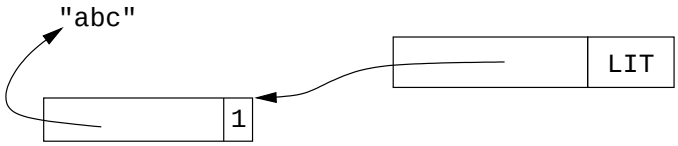


Das erste Wort enthält neben dem Bit, das hier mit dem Wert `0` anzeigt, daß es sich um einen Namen handelt, noch einen Zeiger `home` auf den erzeugenden Berechnungsraum. Das Feld `id` enthält eine eindeutige Zahl, die zum Hashen in Zusammenhang mit Records benötigt wird [Meh99]: zum Hashen kann nicht die Adresse des Namens in der Halde verwandt werden, da dieser sich ja durch die Speicherbereinigung verändern kann.

Auf das `home`-Feld kann im flachen Oz verzichtet werden.

Atome

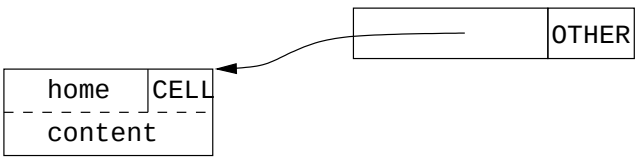
Analog reicht für Atome neben dem Bit, das anzeigt, daß es sich um ein Atom handelt, ein Zeiger auf die Zeichenkette des Atoms:



Durch die gemeinsame Verwendung der Marke `LIT` wird die explizite Unterscheidung in Literale und Atome teurer. Dies kommt aber selten vor: bei Atomen praktisch nur, wenn die Zeichenkette betrachtet wird, was etwa beim Ausdrucken der Fall ist.

7.3.3 Zellen

Zellen werden in Mozart als Objekte mit sekundärer Marke dargestellt:



Neben dem obligatorischen `home`-Zeiger reicht hier ein weiteres Wort `content` für den Inhalt der Zelle, was wieder selbst eine markierte Referenz ist.

In L können das `home`-Feld und die sekundäre Marke wegfallen, so daß hier also lediglich das Feld `content` zur Darstellung einer Zelle ausreicht. Diese Einsparung könnte man auch in Mozart erreichen, wenn man eine eigene Marke für solche Zellen einführen würde, die auf Toplevel erzeugt wurden. Diese Optimierung wurde aber bewußt nicht vorgenommen, da Zellen in der Praxis eine nur untergeordnete Rolle spielen. So werden Zellen zwar zur Konstruktion von Objekten, Locks und den Datentypen Array und Dictionary benötigt; allerdings verwendet die Maschine für diese Datentypen eigene spezielle Datenstrukturen [Hen97, Meh99], so daß eigentliche Zellen nur extrem selten verwandt werden.

7.3.4 Tupel

Bei der Darstellung von Tupeln und Listen (vgl. Abschnitt 4.3.4) folgt Mozart im wesentlichen dem Vorbild der WAM.

Abbildung 7.3 Darstellung von `f(5,[1,2])` im Speicher

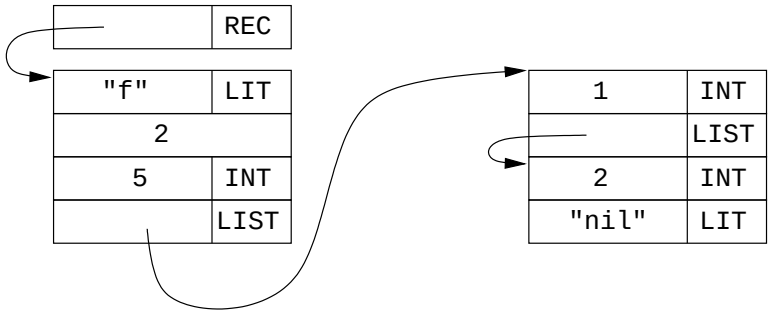
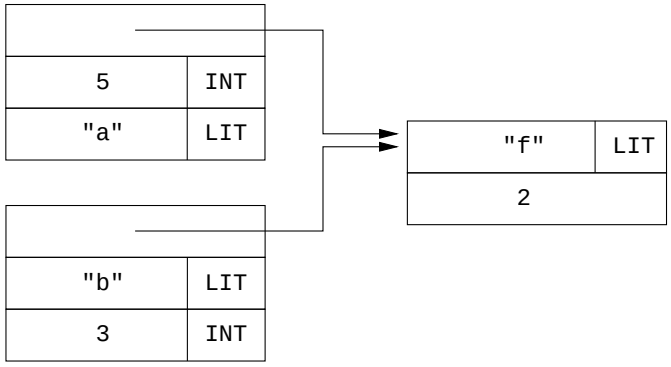


Abbildung 7.3 zeigt die Darstellung des Tupels `f(5,[1,2])` im Speicher. Für die Darstellung eines Listenelementes (markiert mit `LIST`) reichen wie rechts im Bild zu sehen wie in der WAM zwei Worte aus. Dagegen benötigen andere n-stellige Tupel wie links im Bild zu sehen $n + 2$ Worte: je ein Wort für die Marke und die Arität und für jedes Argument ein eigenes Wort. Referenzen auf Tupel werden in Mozart mit `REC` markiert, da es sich bei Tupeln um einen Sonderfall der dort verwandten Records [ST94, RMS96] handelt.

Die WAM verwendet für Marke und Arität nicht jeweils ein eigenes Wort, sondern legt beide Informationen zusammen in einem einzigen Maschinenwort ab. Das bedeutet neben komplexeren Zugriffsfunktionen vor allem auch eine Einschränkung, was die maximale Arität eines Tupels angeht. So erlauben sowohl Sicstus Prolog als auch Quintus Prolog nur maximal 256 Argumente pro Tupel. Anderseits wird durch die kompaktere Darstellung bei der Unifikation lediglich ein Test zum Vergleich von Marke und Arität benötigt. Da nun Tupel in diesen Sprachen oft die Funktion von (einmal schreibbaren) Feldern in imperativen Sprachen übernehmen, schien uns diese Restriktion zu streng, so daß wir an dem etwas mehr Speicherplatz benötigenden Modell festgehalten haben.

Abbildung 7.4 zeigt eine weitere Alternative zur Darstellung von Tupeln, die diesen Nachteil wettmachen kann: es wird (ähnlich wie bei Atomen) sichergestellt, daß ein Paar aus Arität und Marke stets eindeutig im Speicher dargestellt wird, so daß alle Tupel, die sowohl in Marke als

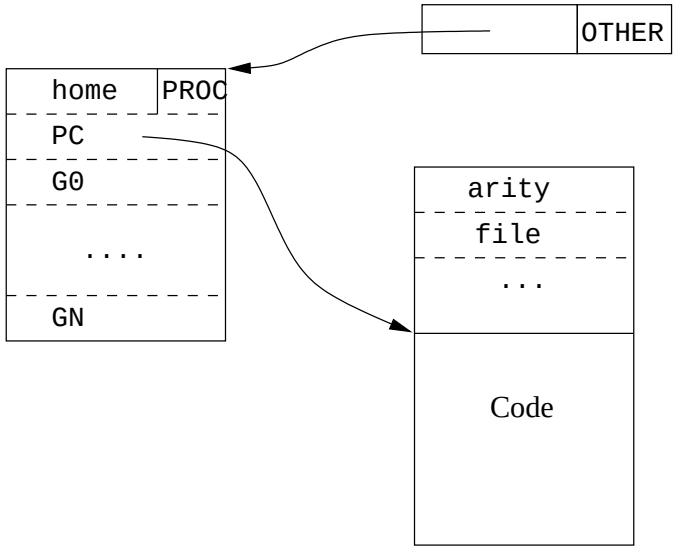
Abbildung 7.4 Sharing von Marke und Arität der beiden Tupel `f(5,a)` und `f(b,3)`



auch Arität übereinstimmen, sich eine Speicherstruktur teilen. Hierbei handelt es sich um die gleiche Idee, die auch bei der effizienten Implementierung von Records [RMS96, Meh99] zur Anwendung kommt. Die Vorteile sind hierbei: Speicherplatzersparnis von einem Wort und damit effizienterer Vergleich auf Gleichheit von Marke und Arität in einem Schritt. Die Nachteile: je ein zusätzlicher Speicherzugriff bei Zugriff auf Marke oder Arität und Mehrkosten beim dynamischen Erzeugen von Tupeln, wenn Marke oder Arität statisch nicht bekannt sind (was allerdings selten vorkommt).

7.3.5 Funktionen

Abbildung 7.5 Darstellung einer Prozedur in Mozart



Zur Darstellung von Funktionen in L wird mindestens benötigt: die Arität, ein Verweis auf den Maschinencode und ein Verweis auf die G-Register. In Mozart wird darüber hinaus noch eine ganze Reihe weiterer Informationen für Funktionen vorgehalten (vgl. Abbildung 7.5): neben dem obligatorischen `home`-Zeiger sind das eine ganze Reihe weiterer Werte, die vorwiegend für das

Debugging benötigt werden, wie zum Beispiel der Printname der Funktion, die Datei samt Zeile, in der diese definiert wurde, etc. Daher wird bei der Darstellung einer Funktionsdefinition in einen dynamischen (die G-Register) und einen statischen Teil unterschieden, wobei nur der dynamische Teil bei jeder Ausführung einer Funktiondefinition neu erzeugt werden muß. Neben dem Maschinencode liegen auch alle diese Zusatzinformationen im statischen Teil, so daß beim Ausführen einer Funktiondefinition dadurch kein Nachteil entsteht. Zudem ist der Speicherbedarf der Zusatzinformationen unbedeutend im Vergleich zur Größe des Codes.

Funktionen verwenden wie Zellen eine sekundäre Marke zur Identifikation ihres Typs. Dies erhöht die Kosten für den Typtest. Allerdings kann in vielen Fällen mit den in Kapitel 4 vorgestellten Techniken bei der Applikation ein Typtest vermieden werden, so daß lediglich bei den wirklichen higher-order Verwendungen von Funktionen Mehrkosten entstehen.

7.4 Variablen

Wir werden im folgenden die Darstellung von Variablen in Mozart betrachten. Wir gehen dabei wie folgt vor:

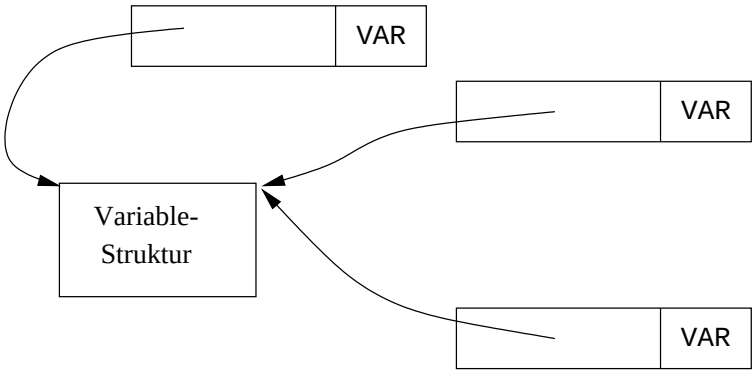
- Abschnitt 7.4.1 geht auf die Problematik des Bindens von Variablen ein, wenn die Halde mehrere Referenzen auf die gleiche Variable enthält.
- Abschnitt 7.4.2 zeigt, daß durch Einführung von markierten Referenzen eine Variable beim Binden an eine Nicht-Variable völlig verschwinden kann.
- Abschnitt 7.4.3 zeigt, daß durch Aufsplittung einer Variable in einen Kopf und einen Rumpf, der Rumpf nach Binden freigegeben werden kann.
- Abschnitt 7.4.4 legt dar, daß die Argumente von Tupeln auch direkt VAR-Zellen enthalten können.
- Abschnitt 7.4.5 geht auf die Problematik von Kellervariablen ein.
- In Abschnitt 7.4.6 zeigen wir, daß in Mozart durch eine optimierte Darstellung von Variablen ohne Suspensionsliste deutlich Speicher und Laufzeit eingespart werden können.
- Abschnitt 7.4.7 vergleicht die Darstellung von Variablen in Mozart mit der in der WAM.
- Abschnitt 7.4.8 gibt mögliche Optimierungen zur Darstellung von Variablen für die flache Teilsprache L an.

7.4.1 Binden von Variablen

Wie bereits eingangs erwähnt, muß durch die Einführung von markierten Referenzen statt markierter Objekte das Binden von Variablen besonders behandelt werden, da Variablen die einzigen Halden-Zellen sind, deren Marke sich ändern kann.

Abbildung 7.6 zeigt eines Situation, bei der die Halde mehrere Referenzen auf eine Variable enthält. Wird diese Variable nun zum Beispiel an eine Zahl gebunden, so sind unterschiedliche Vorgehensweisen möglich. So könnte man etwa *alle* diese VAR-Zellen ändern. Dies ist allerdings nicht ohne großen Aufwand (entweder in Speicherplatz oder Laufzeit) zu realisieren.

Abbildung 7.6 Mehrere Referenzen auf dieselbe Variable



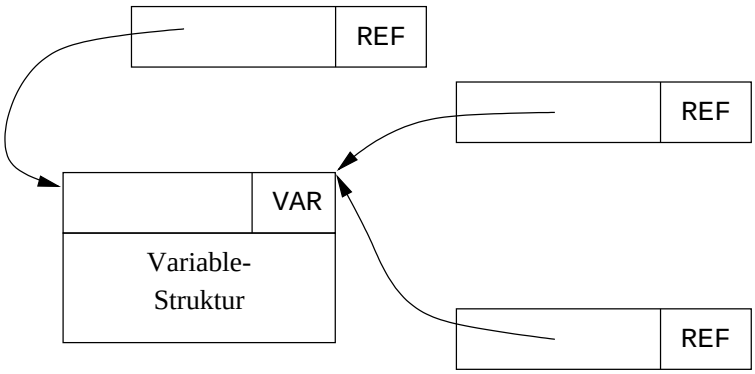
Eine andere Möglichkeit wäre die, in der Variable-Struktur mittels REF zu vermerken, ob die Variable gebunden ist. Trifft man dann beim Dereferenzieren auf eine VAR-Zelle, so folgt man dem Zeiger auf die Variable-Struktur, prüft dort, ob diese gebunden ist und fährt gegebenenfalls weiter so fort. Dies verteuert aber die häufig verwandte Dereferenzierungsoperation deutlich, da neben dem Test auf VAR jeweils noch das Nachschauen in der Variable-Struktur benötigt wird.

Abbildung 7.7 zeigt eine Darstellung, die diesen Nachteil nicht hat. Die Idee dabei ist, daß man die Marke `VAR` bei Variablen gerade nicht aus dem Objekt herauslagert, sondern in der Variable selbst beläßt. Verweise auf die Variable sind dann alle mit `REF` markiert. Dies führt zu folgender Invariante:

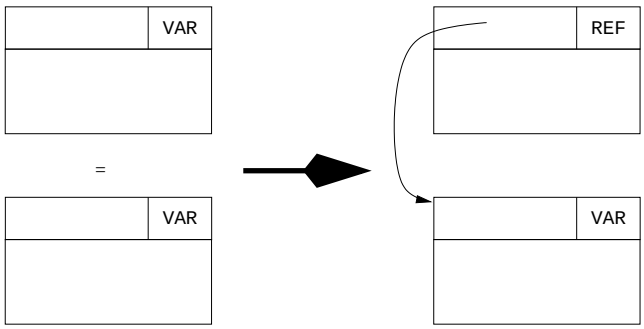
Invariante 1 *Zu jeder Variablen gibt es genau eine Speicherzelle auf der Halde, die mit* `VAR` *markiert ist.*

Die Einhaltung der Invariante im allgemeinen ist nun leichter, als es auf den ersten Blick scheinen mag, da Werte in der Regel ohne weitere Dereferenzierungsschritte in der Maschine weitergereicht werden. Da die Identität einer Variable durch die Lage in der Halde und nicht durch den Inhalt dieser Haldenzelle definiert ist, muß sichergestellt werden, daß niemals der Inhalt einer Variablen-Zelle an eine andere Stelle kopiert wird. Dazu wird sichergestellt, daß Register nie Variablen, sondern nur `REF`-Zellen enthalten können:

Abbildung 7.7 Mehrere Referenzen auf dieselbe Variable unter Verwendung von REF-Zellen



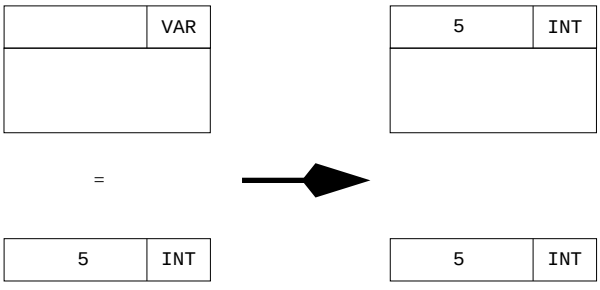
- Beim Erzeugen einer neuen Variable mittels `newVar` R auf der Halde eine neue variable-Struktur mit `VAR`-Zelle alloziert. In das zugehörige Register R wird nur eine `REF`-Zelle auf diese Variable abgelegt.
- Anders als beim Binden einer Variable an eine Nichtvariable (vgl. Abschnitt 7.4.2), muß bei der Unifikation zweier Variablen eine der beiden zu einer `REF`-Zelle geändert werden:



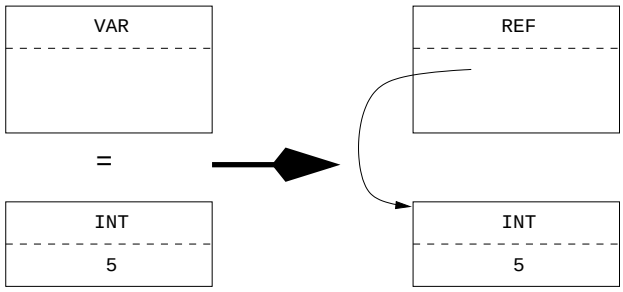
Dies sind bereits alle Stellen, die zur Einhaltung der Invariante zu berücksichtigen sind. Eine einzige weitere Stelle kommt in Betracht, wenn man `VAR`-Zellen direkt in Argumentposition von Tupeln erlaubt. Wir kommen darauf in Abschnitt 7.4.4 zurück.

7.4.2 Binden an Nicht-Variablen

Man beachte, daß nur beim Binden zweier Variablen aneinander eine neue REF-Zelle entsteht. Wenn eine Variable dagegen mit einer Nicht-Variable unifiziert wird, kann die Variable-Zelle direkt mit dem Inhalt der anderen Zelle überschrieben werden. So wird in folgendem Beispiel, bei der Unifikation mit einer Zahl, die Variable-Zelle mit dem Inhalt der Zahl-Zelle überschrieben:

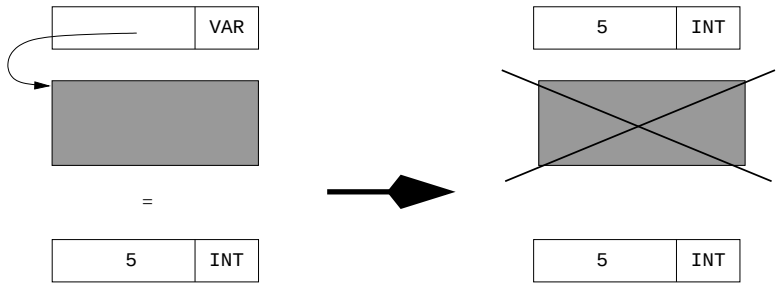


Nach der Unifikation kann nicht mehr erkannt werden, daß die Zahl-Zelle ursprünglich eine Variable enthielt. Dies ist eine wichtige Eigenschaft, die durch die Einführung markierter Referenzen bewirkt wird. Bei der Darstellung über markierte Objekte muß dagegen stets eine neue REF-Zelle eingeführt werden. Obiges Beispiel würde also bei Darstellung über markierte Objekte wie folgt aussehen:



7.4.3 Aufsplittung

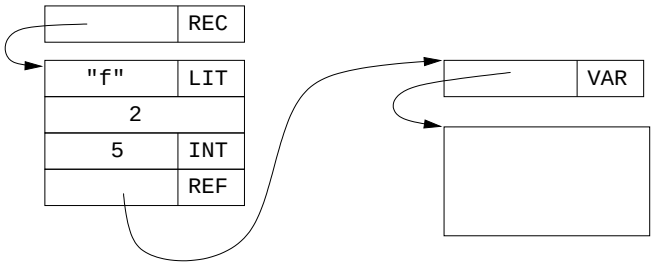
Nach dem Binden einer Variablen (außerhalb eines Wächters) wird von der Variable-Struktur nur noch die zuvor mit `VAR` markierte Zelle benötigt. Der Rest der Struktur kann freigegeben und wiederverwandt werden. Wir splitten daher die Darstellung einer Variable in einen Kopf und einen Rumpf auf:



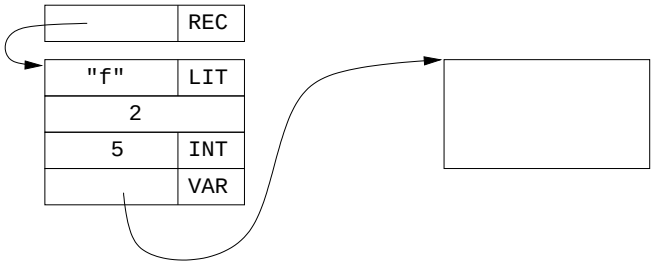
Beim Binden kann dann der Rumpf freigegeben und über die Freispeicherliste wiederverwandt werden, da ja alle Verweise auf die Variable auf deren Kopf und nie direkt auf deren Rumpf zeigen.

7.4.4 Variablen in Argumenten von Tupeln

Kommt eine Variable in einem Argument eines Tupels vor, dann wird, wie bisher beschrieben, zunächst eine neue Variable auf der Halde erzeugt und dann eine Referenz darauf im entsprechenden Argument des Tupels eingetragen, so daß folgende Situation entsteht:



Mit der Einführung markierter Referenzen, ist es aber sinnvoller, den Kopf der Variablen direkt im Argument des Tupels selbst abzulegen:



Dadurch spart man Speicher (und damit auch Laufzeit) für die separate Allokation der Variablen. Zudem verschwindet beim Binden dieser Variablen jeder Hinweis auf ihre Existenz; in der bisherigen Variante wären hier in Argumentposition REF-Zellen zurückgeblieben, die erst bei der Speicherbereinigung beseitigt werden können.

Damit die Maschine direkt in Argumentposition Variablen erzeugen kann, muß eine neue Instruktion `setVarArg` eingeführt werden:

`setVarArg` R
: Erzeugt in dem Argument eines Tupels, auf den der Argumentzeiger `ap` verweist, eine neue Variable-Zelle. Im Register R wird eine REF-Zelle abgelegt, die auf dieses Argument verweist. Anschließend wird `ap` inkrementiert.

Analog dazu muß auch die duale Instruktion `getVarArg` für die Lese-/Schreibunifikation (die im Verbund mit `unifyArg` und `moveArg` eingesetzt wird, vgl. Abschnitt 4.3.2) definiert werden:

`getVarArg` R
: Wenn `mode` den Wert `write` hat, arbeitet die Instruktion genau wie `setVarArg`. Andernfalls wird das Argument, auf das `ap` verweist, in R geladen: handelt es sich dabei um eine Variable-Zelle, wird zuerst in R eine REF-Zelle darauf erzeugt.

Hier ist also zu beachten, daß Invariante 1 erhalten bleibt. Dazu muß beim Laden eines Argumentes eines Tupels nun zusätzlich geprüft werden, ob es sich um eine VAR-Zelle handelt und daher gegebenenfalls zuerst eine REF-Zelle erzeugt wird.

7.4.5 Kellervariablen

Das Erzeugen einer neuen `VAR`-Zelle für lokale Bezeichner einer Funktion geschieht wie zuvor beschrieben unter Verwendung der Instruktion `newVar` $\mathtt{L}_x$. Diese alloziert eine neue `VAR`-Zelle auf der Halde und legt im Register $\mathtt{L}_x$ eine `REF`-Zelle mit Referenz auf diese Variable ab. Auch nach Binden von x (beispielsweise an eine Zahl) enthält $\mathtt{L}_x$ immer noch diese `REF` Zelle, die frühestens bei der Speicherbereinigung beseitigt werden kann. Auch der für die Variable allozierte Speicherbereich kann erst wieder durch die Speicherbereinigung freigegeben werden.

Bei Mozart ist der Anteil dieser Kellervariablen besonders hoch, weil diese ja zur Realisierung der Ausgabewerte von funktionalen Prozeduren erzeugt werden.

Die WAM geht beim Erzeugen neuer Variablen intelligenter vor: hier würde die `VAR`-Zelle für x direkt im L-Registersatz gespeichert, weshalb wir im folgenden von Kellervariablen sprechen wollen. Man spart so (unter Umständen nur kurzzeitig, siehe unten) die Allokation von Speicher auf der Halde. Auch nach dem Binden von x sehen spätere Zugriffe auf x mittels $\mathtt{L}_i$ nun keine

REF-Zelle mehr, gerade so als hätte nie eine VAR-Zelle existiert. Somit entstehen in Oz mehr Referenz-Ketten der Länge 1 als in Prolog (vgl. auch Abschnitt 5.3).

Das Vorgehen der WAM bringt jedoch auch eine ganze Reihe von deutlichen Komplikationen mit sich. Da die L-Register nach dem Verlassen einer Funktion auch wieder freigegeben werden, muß sichergestellt werden, daß nach Deallokation der L-Register keine Referenzen mehr auf solche Variablen existieren. In der WAM muß daher bei der Unifikation einer Halden-Variable mit einer Keller-Variablen die Keller-Variable präferiert gebunden werden. Auch bei der Unifikation zweier Keller-Variablen muß stets die zuletzt erzeugte an die ältere gebunden werden. Dies macht es erforderlich, daß einfach festzustellen ist, ob eine Variable auf dem Keller oder der Halde liegt; in Sicstus Prolog wurden daher zur Unterscheidung eigene Marken für Halden- und Kellervariablen eingeführt [Car91]. Schließlich wird auch der Compiler verkompliziert; das letzte Vorkommen einer Kellervariablen im Rumpf einer Funktion muß nämlich gesondert behandelt werden: besondere `unsafe`-Varianten der entsprechenden Maschineninstruktionen müssen in der WAM sicherstellen, daß eine dann noch nicht gebundene Kellervariable *globalisiert*, d.h.auf die Halde kopiert wird (wodurch der Vorteil der Allokation im Keller wieder verloren geht).

In Oz kommt ein zusätzliche Komplikation durch die Nebenläufigkeit hinzu: werden zwei Kellervariablen unifiziert, so kann es zu einer Referenz eines L-Registersatzes in einen anderen Satz kommen, was nicht sein darf, da nicht sicher ist, welche L-Register zuerst deallociert werden wird, wenn diese in verschiedenen Threads liegen. In einem solchen Fall müßte eine dritte Variable auf der Halde erzeugt werden, an die dann die beiden Kellervariablen gebunden werden.

Wegen ihrer Komplexität wurden in Mozart vergleichbare Techniken noch nicht integriert. Dennoch ist es eine lohnende Investition, den Speicherverbrauch von Kellervariablen zu reduzieren, wie Tabelle 10.12 auf Seite 205 verdeutlicht. Die Tabelle zeigt unter anderem den Anteil der oben diskutierten Kellervariablen am Gesamtspeicherverbrauch verschiedener Applikationen. Hier liegt ein beträchtliches Einsparpotential, das bis zu einem Drittel des verbrauchten Haldenspeichers betragen kann.

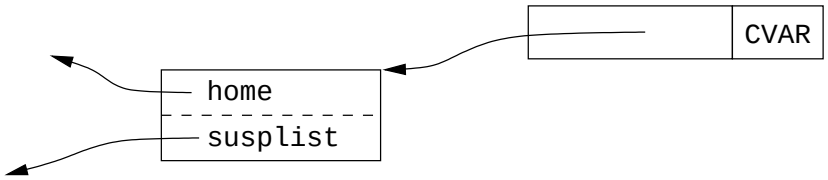
Auch hochperformante Systeme sind dazu übergegangen, Variablen grundsätzlich nicht auf dem Keller zu allozieren [VR90]. Bei Aquarius ist diese Entscheidung allerdings gut vertretbar, wenn man sie im Zusammenhang mit anderen Optimierungen betrachtet, die bestimmte Fälle mit den Techniken der funktionalen Programmiersprachen übersetzen: so wird hier durch statische Analyse erkannt, welche Argumente einer Funktion als Ausgabeargumente dienen. Dann werden beim Aufruf spezielle Varianten der Funktion angesprungen, so daß erst gar keine Variablen erzeugt werden müssen.

7.4.6 Variablen in Mozart

Aus Optimierungsgründen werden in Mozart Variablen auf zwei unterschiedliche Arten dargestellt.

Constraint-Variablen

In Mozart werden Variablen unter Verwendung der Marke `CVAR` (= Variablen mit Constraints, näheres weiter unten) und einem Zeiger auf ein Objekt der C++-Klasse `Variable` dargestellt:



Die Klasse `Variable` enthält neben der Referenz `home`, die auf den die Variable erzeugenden Berechnungsraum verweist, eine Referenz `susplist` auf die Suspensionsliste der Variable. Dabei handelt es sich, wie in Abschnitt 3.7.1 auf Seite 54 bereits ausführlich besprochen, um die Liste derjenigen Threads, die auf die Variable suspendieren.

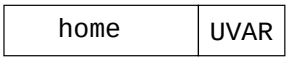
Durch Bildung von Unterklassen von `Variable` kann die Maschine um weitere spezielle Arten von Variablen erweitert werden, mit denen noch mehr Information assoziiert werden kann. Beispiele für solche Variablen sind Variablen über endlichen Domänen (Finite Domains) [Wür98], offenen Records [RMS96], endlichen Mengen, Lazy-Variablen (vgl. Abschnitt 5.7). Auch zur Darstellung von Variablen im verteilten Oz wurde dieser Datentyp herangezogen [VHB+97].

Die Integration von Constraint-Variablen in die Maschine wurde generisch ausgelegt, so daß eine Erweiterung um zusätzliche Domänen sehr einfach vorgenommen werden kann: die Integration einer neuen Art einer Constraint-Variable muß lediglich eigene Routinen zur Unifikation, zum Binden, zum Wecken und zur Speicherbereinigung bereitstellen. Alle entsprechenden Stellen in der Maschine wurden herausfaktorisiert und entsprechend modifiziert, so daß diese Routinen bei Bedarf aufgerufen werden.

Variablen ohne Suspensionsliste

Wie bereits beschrieben, müssen beim Binden einer Variable alle Threads in der Suspensionliste geweckt werden. Ausgehend von der mit `CVAR` markierten Referenz muß dabei also im Detail folgendes geschehen: zunächst muß der Zeiger auf die Variable-Struktur beschafft werden, danach muß das Feld `susplist` aus dem Speicher geladen und anschließend getestet werden, ob es leer ist. Das sind vergleichsweise teuere Operationen. Zudem ist das Binden einer Variable in Mozart eine der häufigsten und zentralsten Operationen der Maschine, vor allem weil Mozart keine echten Funktionen bietet, sondern diese durch das Erzeugen und Binden von Variablen realisiert werden müssen. Zudem existieren nur sehr wenige Variablen mit nicht-leerer Suspensionliste.

Um das Binden zu Beschleunigen und um Speicher zu sparen (und damit auch das Erzeugen einer Variable zu beschleunigen) verwendet Mozart für Variablen mit leerer Suspensionsliste eine spezielle Darstellung:



Hier enthält das Referenzfeld der mit `UVAR` (= unconstrained variable) markierten Referenz einen direkten Verweis auf den erzeugenden `home`-Zeiger, so daß die Allokation einer separaten Struktur für den Rumpf der Variable entfallen kann. Diese Struktur wird erst dann alloziert, wenn der (seltene) Fall eintritt, daß ein Thread auf eine solche Variable suspendiert: dann wird die Marke der Haldenzelle direkt in `CVAR` geändert, und der `home`-Zeiger wird durch einen Verweis auf die gerade allozierte Struktur ersetzt.

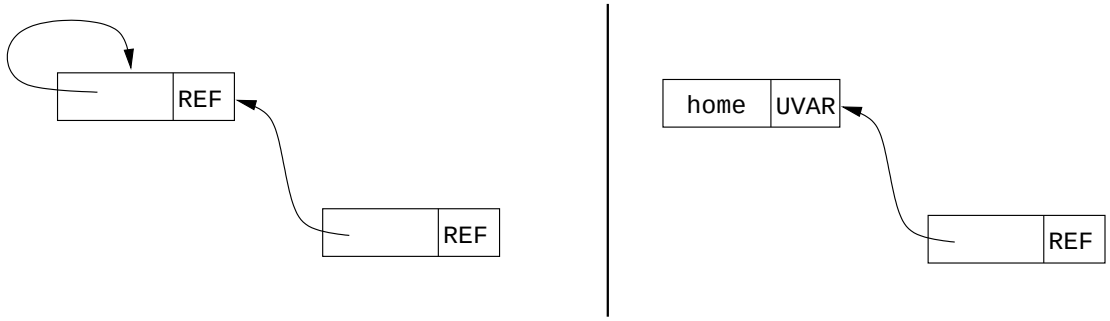
Beim Binden einer `UVAR` muß nun die Suspensionliste nicht mehr betrachtet werden. In Mozart

muß noch geprüft werden, ob `home` gleich dem aktuellen Berechnungsraum ist, was in der Praxis fast immer der Fall ist. Daher wird hier in Mozart weiter optimiert: alle `UVAR` Variablen, die im gleichen Berechnungsraum erzeugt wurden sehen auch alle gleich aus, da ja sowohl Marke (`UVAR`) als auch Referenzteil (`home`) übereinstimmen; die Identität dieser Variablen ist lediglich durch die Lage in der Halde bestimmt (siehe unten). Daher verwendet die Maschine ein spezielles Register `curUVARPrototye`, das gerade den Prototyp einer UVAR-Variablen des aktuellen Berechnungsraumes enthält. Dieses Register wird bei jeder Änderung des Berechnungsraumes angepaßt. Es wird zum einen beim Erzeugen einer neuen Variablen auf der Halde verwandt: ein Wort wird auf der Halde alloziert und der Inhalt dieses Wortes wird auf den Wert von `curUVARPrototye` gesetzt, so daß eine Shift- und Markierungsoperation entfallen kann. Zudem wird das Binden unter Verwendung dieses Registers beschleunigt: wenn die zu bindende Variable dem Inhalt von `curUVARPrototye` entspricht, kann die entsprechende Speicherzelle direkt ohne weitere Aktionen überschrieben werden. Das Binden einer Variable ist also in fast allen Fällen mit einem einzigen Test (neben dem Dereferenzieren) verbunden.

7.4.7 Variablen in der WAM

Abbildung 7.8 zeigt vergleichend die beiden Darstellungsvarianten für Variablen, links die in Prolog übliche Darstellung, rechts in Oz. In Prolog werden ungebundene Variablen durch `REF` Zellen realisiert, die auf sich selbst verweisen. Diese Darstellung ist deshalb möglich, weil Variablen in Prolog sonst keine weitere Information tragen.

Abbildung 7.8 Darstellung von Variablen: links in Prolog, rechts in Mozart.



Der Vorteil der Prolog-Variante liegt zum einen darin, daß für Variablen lediglich eine Marke `REF` verwendet wird, während in Mozart gebundene und ungebunden Zellen unterschieden werden müssen. Zum anderen kann man in Prolog den Inhalt einer Speicherzelle unbesehen in eine andere Speicherzelle kopieren: der Zeiger einer `REF`-Zelle wandert hier quasi automatisch richtig mit. In Mozart muß man dagegen an den entsprechenden Stellen (die nur selten auftreten, wie wir oben gesehen haben) darauf Rücksicht nehmen und zusätzliche Tests vorsehen.

Vorteile hinsichtlich der Länge von Referenzketten und beim Dereferenzieren selbst ergeben sich in der Prolog-Darstellung nicht, da ja auch hier solche Ketten stets in einem Register mit einer `REF` Zelle beginnen müssen.

7.4.8 Variablen in L

In der flachen Teilsprache L entfällt die Notwendigkeit für jede Variable den Berechnungsraum zu merken, in dem diese erzeugt wurde. Gegenüber Prolog muß aber zusätzlich die Suspensionsliste einer Variable verwaltet werden. Man könnte daher in L eine Unterscheidung vornehmen in Variablen mit und ohne Suspensionsliste. Für Variablen ohne Suspensionsliste kann man dann auf die Darstellung der WAM zurückgreifen (selbstreferenzierende REF-Zelle), während man für diejenigen mit Suspensionen analog zu den CVAR-Variablen aus Abschnitt 7.4.6 vorgehen kann. Dann muß man aber auch wieder beim Laden von CVAR-Variablen in Register darauf achten, daß man nur Referenzen auf diese lädt, was für die REF-Variablen nicht gelten würde. Will man daher gänzlich diesen zusätzlichen Test in der flachen Maschine einsparen, so muß man nur darauf achten, daß CVAR-Variablen nicht in Argumenten von Tupeln und Listen auftreten, daß heißt beim Umwandeln einer REF-Variable in eine CVAR-Variable alloziert man eine neue CVAR-Zelle auf der Halde und bindet die REF-Zelle daran.

7.5 Rationale Unifikation

Da rationale Bäume durch Graphen mit Zyklen dargestellt werden, muß darauf bei der Unifikation besonders geachtet werden: Quintus Prolog kommt beispielsweise nicht mit allen zyklischen Termen zurecht und läuft unter Umständen in eine Endlosschleife, während Sicstus Prolog die rationale Unifikation korrekt durchführt.

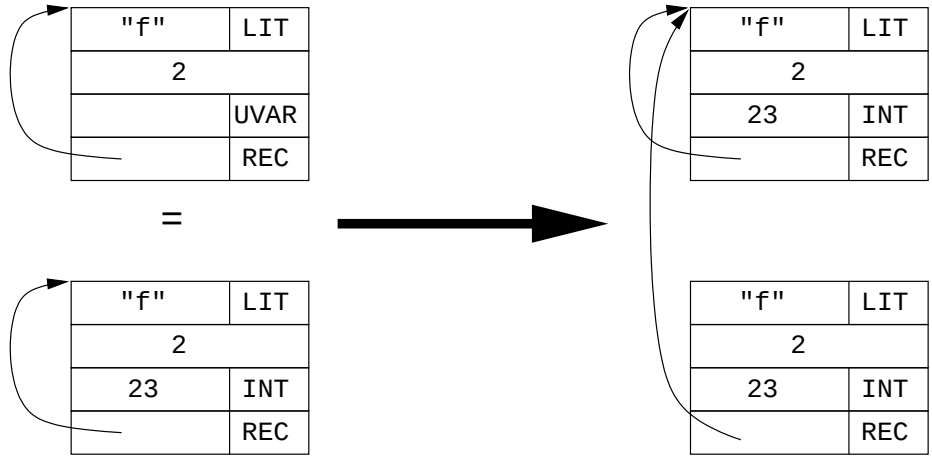
Zum Erkennen von Zyklen geht man abstrakt gesehen so vor, daß man eine Tabelle verwaltet, in der man alle während einer Unifikation besuchten Tupelpaare $\langle t, t' \rangle$ einträgt. Wird nun während des rekursiven Abstiegs ein Paar besucht, das bereits in der Tabelle vorhanden ist, so braucht dieser Pfad nicht weiter verfolgt werden. Eine explizite Repräsentation der Tabelle kommt meist in parallelen Implementierungen zum Einsatz, da hier mehrere Prozessoren (*worker*) gleichzeitig an denselben Strukturen arbeiten können. In der sequentiellen Implementierung Mozart dagegen wird die Tabelle implizit durch eine temporäre (für die Dauer der Unifikation) Modifikation der Strukturen dargestellt: beim rekursiven Abstieg in zwei Tupel t und t' wird jeweils der Wert eines Argumentes der Tupel durch das korrespondierde Argument des anderen Tupels ersetzt.

Abbildung 7.9 zeigt das Vorgehen am Beispiel der Unifikation zweier zweistelliger Tupel, die im zweiten Argument jeweils eine Referenz auf sich selbst enthalten. Wenn bei der Unifikation der Argumente zweier Tupel wieder rekursiv in zwei Tupel abgestiegen wird, dann wird zuerst eine Strukturbindung vorgenommen, das heißt eine der beiden REC-Zellen wird durch die andere ersetzt. Definiert man eine totale Ordnung auf REC-Zellen (z.B. über die Adresse der Tupel in der Halde, auf die sie verweisen) und ersetzt stets größere durch kleinere REC-Zellen, dann muß die Unifikation terminieren, da eine Rekursion nur über REC-Zellen möglich ist und ihre Menge a priori endlich ist.

7.6 Lokale Variablen im Case

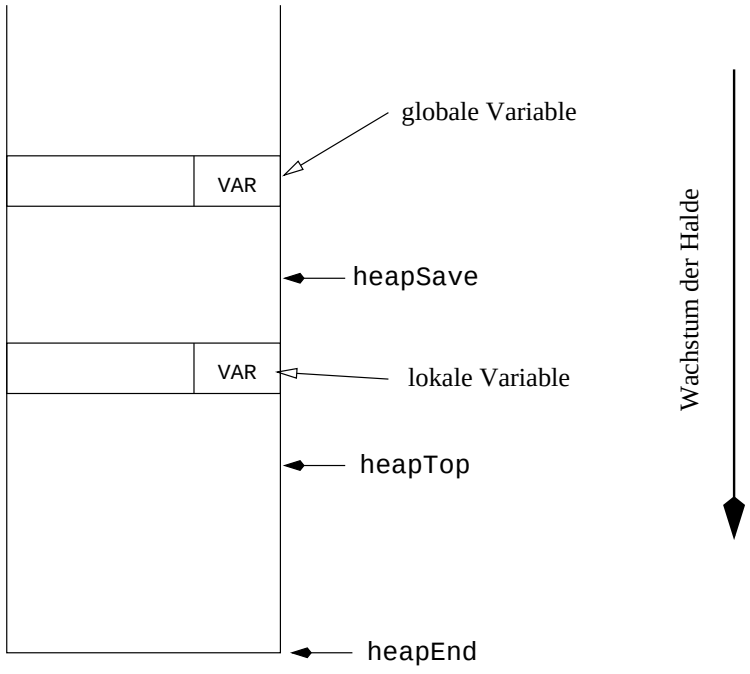
Für die Entscheidung von Subsumption eines Wächters, ist es wichtig, zu erkennen, ob eine logische Variable innnerhalb dieses Wächters erzeugt wurde. Aus Abschnitt 3.9 sind wir noch eine Erklärung schuldig geblieben, die beschreibt, wie die Maschine diese Bedingung einfach entscheiden kann. Das wollen wir jetzt nachholen.

Abbildung 7.9 Rationale Unifikation zweier zyklischer Tupel.



Die Maschine verwendet hierzu das Register `heapSave`, das ein Zeiger in die Halde ist und normalerweise auf den Anfang der Halde zeigt, also auf die Adresse 0. Wie wir in Abschnitt 3.9 gesehen haben, wird der Code eines Wächters durch eine Instruktion `guardStart` eingeleitet. Die Instruktion setzt nun zusätzlich `heapSave` auf den aktuellen Wert von `heapTop`. Beim Binden einer Variable v innerhalb eines Wächters kann nun sehr einfach entschieden werden ob diese lokal ist (vgl. Abbildung 7.10): v ist genau dann lokal, wenn v unterhalb (bei Wachstum der Halde in Richtung kleinerer Adressen, vgl. Abschnitt 7.1.1) von `heapSave` in der Halde liegt. Es reicht

Abbildung 7.10 Das Register `heapSave` zur Unterscheidung von lokalen und globalen Variablen.



also ein einfacher Vergleich zweier Adressen.

Bei Verlassen eines Wächters wegen Subsumption, Dissupsumption oder Suspension muß darauf geachtet werden, daß `heapSave` wieder auf die Adresse 0 zurückgesetzt wird. Das hat einen weiteren Vorteil: beim Binden einer Variable außerhalb eines Wächters werden alle Variablen automatisch als lokal behandelt (da ihre Adresse ja stets größer als 0 ist), also wie gewünscht keine Einträge auf der Spur vorgenommen. Dadurch können dann alle Instruktionen, die potentiell Variablen binden können, sowohl außerhalb als auch innerhalb eines Wächters verwandt werden, ohne daß man zwei verschiedene Varianten dieser Instruktionen implementieren müßte.

Zusammenfassung

- Datenstrukturen wie Variablen, Threads, Suspensionslisten und L-Register, bei denen das Ende der Lebensdauer eindeutig bestimmt werden kann, werden über eine *Freispeicherliste* verwaltet.
  Allokation über die Freispeicherliste ist in der Praxis nicht teurer als eine Allokation direkt von der Halde.
  Die Verwendung der Freispeicherliste kann den Speicherverbrauch von realen Applikation drastisch reduzieren.

- Die Typen der Werte werden durch die Verwendung von *markierten Referenzen* kodiert. Durch eine ausgefeilte Darstellung wird erreicht, daß

  – trotz 4 Tag-Bits nur 2 Bits des Adreßraumes verloren gehen,

  – Typtest und Demaskierung für die häufig verwandten Typen besonders effizient geschehen kann

  – das Dereferenzieren effizient geschehen kann

- *Kleine ganze Zahlen* passen in ein Register, so daß diese nicht auf der Halde alloziert werden müssen.

- *Fließkommazahlen* müssen auf der Halde alloziert werden.

- In Mozart werden *Variablen* meist zur Realisierung funktionaler Prozeduren benötigt.

  – Variablen sind durch eine eindeutige Speicherzelle auf der Halde repräsentiert. Diese Zelle darf nicht in Register kopiert werden, es muß stattdessen eine `REF`-Zelle im Register erzeugt werden. Dies ist beim Laden von Argumenten von Tupeln und bei der Unifikation zweier Variablen zu beachten.

  – Bei der Darstellung von Variablen wird unterschieden in Variablen mit und ohne Suspensionsliste. Dadurch kann das Binden von Variablen ohne Suspensionsliste sehr effizient erfolgen.

  – Das Fehlen echter Funktionen in Mozart bedingt einen mitunter deutlich erhöhten Speicherverbrauch durch das Erzeugen von Variablen.

- Lokale Variablen im Wächter eines *Case* können durch einen Adreßvergleich mit dem Register `heapSave` effizient erkannt werden. Dies ist wichtig für die Entscheidung von Subsumption.

Kapitel 8

Threads

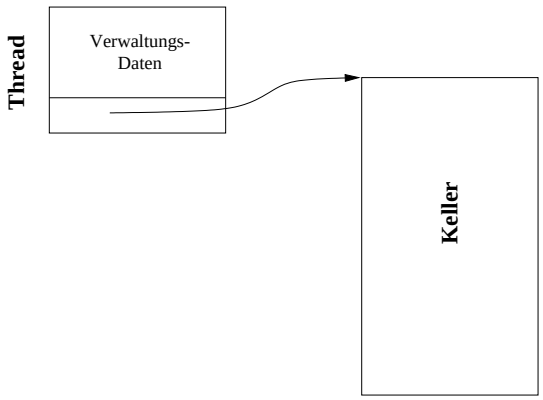
Die Unterstützung nebenläufiger Berechnung ist eine der zentralen Spracheigenschaften von L. In diesem Abschnitt wollen wir zeigen, daß die Bereitstellung von Threads nicht auch gleichzeitig mit Effizienzeinbußen bezahlt werden muß. So sollen nicht-nebenläufige Anwendungen, die rein sequentiell arbeiten, keinen Preis bezahlen; andererseits ist die Erzeugung und Verwaltung von vielen Threads vergleichsweise billig, so daß auch deren intensive Nutzung nicht zu teuer wird. Um Threads effizient implementieren zu können, müssen verschiedene Verfeinerungen und Änderungen gegenüber dem Modell aus Teil II vorgenommen werden, die wir im folgenden beschreiben wollen. Die meisten, der im folgenden vorgestellten Techniken auf Threadebene, können aber auch in einem nicht-nebenläufigen System mit nur einem Berechnungsfaden angewandt werden. Wir werden zunächst in Abschnitt 8.1 kurz auf die Darstellung eines Threads im Speicher eingehen. In Abschnitt 8.2 werden wir sehen, wie wir jeweils den obersten Auftrag eines Threads in der Maschine cachen können. Dann zeigen wir in den Abschnitten 8.3 und 8.4, daß der Test auf Preemption mit dem Test auf Signalbehandlung gekoppelt werden kann und somit keine Extrakosten verursacht. Abschnitt 8.5 beschreibt eine wichtige jedoch transparente Abweichung bei der Implementierung: die X-Register werden im Emulator nicht pro Thread sondern einmal für den ganzen Emulator angelegt. Abschnitt 8.6 zeigt wie man durch die Verwendung spezieller Auftragsarten gleichzeitig an Effizienz aber auch deutlich an Flexibilität gewinnen kann. Wir schließen das Kapitel ab, indem wir in Abschnitt 8.7 eine Diskussion von Threads unter dem Gesichtspunkt der Effizienz führen.

8.1 Darstellung im Speicher

Wie bereits beschrieben, ist ein Thread ein Stapel von Aufträgen. Die Größe des Stapels soll nicht statisch festliegen, sondern dynamisch veränderbar sein: da oft sehr viele Threads gleichzeitig im Speicher existieren, die zudem mit sehr kleinen Stapeln auskommen, kann man dadurch sowohl Laufzeit als auch vor allem deutlich Speicherplatz sparen. Da zu einem Zeitpunkt allerdings viele Referenzen auf einen Thread existieren können (z.B. aus den Suspensionslisten verschiedener Variablen heraus), müßten beim Vergrößern, also bei der Reallokation des Stapels, alle diese Referenzen angepaßt werden, was unpraktikabel ist. In Mozart stellen wir daher Threads durch Einführung einer Indirektion so dar, wie es in Abbildung 8.1 verdeutlicht wird.

Die Indirektion bedeutet keine Einbußen beim effizienten Zugriff auf den Stapel: wird ein Thread zur Ausführung gebracht, so zeigt das Register `SP` nach wie vor direkt auf den obersten Eintrag

Abbildung 8.1 Speicherrepräsentation eines Threads



des Kellers.

Der Bereich „Verwaltungs-Daten“ enthält Informationen wie die Priorität des Threads, eine eindeutige Zahl zur Identifikation des Threads, Flags (z.B. für Debugging), einen Zeiger für die Verkettung in der Schlange `runnable`, etc. Da viele Threads mit einem kleinen Stapel auskommen, wird bei der ersten Allokation eines Threads die initiale Größe des Stapels klein (= wenige Worte) gewählt, und erst beim ersten Vergrößern deutlich größer angelegt. Dadurch wird erreicht, daß die Erzeugung eines Threads billig ist (gemessen in Laufzeit und Speicherplatz).

8.2 Cachen des obersten Auftrages

Wie in Abschnitt 3.5.5 beschrieben interpretiert der Emulator die Maschineninstruktionen stets relativ zur aktuellen Umgebung, die durch den obersten Auftrag des ausführenden Threads bestimmt wird. Dieser enthält den Programmzeiger, und je einen Zeiger auf den aktuellen L- und G-Registersatz. Da diese Zeiger sehr häufig zugegriffen und geändert werden (vor allem der Programmzähler), ist die Realisierung eines effizienten Zugriffs sehr wichtig für die Ausführungsgeschwindigkeit des Emulators. Der Emulator benutzt daher drei zusätzliche Register `PC`, `L` und `G`, in denen er den obersten Auftrag cached. Der Emulator muß dazu entsprechend an den folgenden Stellen modifiziert werden:

- Wenn ein Thread T in den Zustand laufend wechselt, wird der oberste Auftrag von T entfernt und `PC`, `L`, `G` mit den Zeigern dieses Auftrages initialisiert.
- Bei der Applikation einer Funktion f werden die aktuellen Werte von `PC`, `L`, `G` auf dem ausführenden Thread gerettet und danach gemäß f neu gesetzt.
- Beim Rücksprung aus einer Funktion wird der oberste Auftrag vom Thread genommen und `PC`, `L`, `G` anhand dieses Auftrages neu gesetzt.
- Bei Suspension und Preemption eines Threads T werden `PC`, `L`, `G` auf T gesichert.

8.3 Fairneß

In Abschnitt 3.6 haben wir gesehen, daß die Zeitscheibe zur Realisierung der Fairneß über eine Prozedur realisiert wird, die das Betriebssystem in regelmäßigen Abständen aufruft. Diese setzt dann lediglich ein Flag, das bei jeder Funktionsapplikation getestet wird. In der Praxis hat sich dies allerdings als nicht ausreichend erwiesen. Grund hierfür war die Verwendung von *links*rekursiven Funktionen. Wir wollen das Problem am Beispiel einer (naiven) Implementierung der Funktion `length` zur Berechnung der Länge einer Liste verdeutlichen:

```
fun length(l) =
  case l of
    h::r => length(r)+1
  | nil  => 0
```

Da die Funktion linksrekursiv ist, bleibt beim rekursiven Abstieg auf dem Keller des ausführenden Threads jeweils noch eine Operatorapplikation (Addition der Zahl 1) zurück. Da bei der Operatorapplikation aus Effizienzgründen kein Test auf Preemption durchgeführt wird, wird also bei sehr langen Listen extrem viel Arbeit auf dem Keller angehäuft, deren Ausführung nicht unterbrechbar ist. In der Praxis traten daher Extremfälle auf, wo ein Thread erst nach einigen CPU Sekunden die Kontrolle wieder an den Scheduler zurückgab, was aus Benutzersicht inakzeptabel ist.

Daher führt der Emulator noch an einer weiteren Stelle einen Test auf Preemption durch, nämlich bei Ausführung der Instruktion `return`. In der Praxis hat sich dieses Vorgehen nun als ausreichend erwiesen, um dem Benutzer die Illusion der zeitgleichen Ausführung von Threads zu bieten.

8.4 Signalbehandlung

Wie in Abschnitt 3.6 beschrieben muß in regelmäßigen Abschnitten ein Test auf die Notwendigkeit einer Signalbehandlung durchgeführt werden. Ein solches Signal kann anzeigen, daß Preemption oder eine Speicherbereinigung nötig ist, daß an den Ein/Ausgabekanälen neue Information anliegt, daß das System im Debugmodus arbeitet und daher zusätzliche Aktionen zu erfolgen haben, etc. Ungünstig wäre es, hier für jedes einzelne Ereignis ein eigenes Register zu verwenden, da dann bei jeder Funktionsapplikation nacheinander alle diese Register getestet werden müßten.

Günstiger ist es, wenn man dem Erfülltsein jeder einzelnen Bedingung ein ganz bestimmtes Bit innerhalb eines einzigen Bitvektors zuweist: das Bit ist genau dann gesetzt, wenn die Bedingung (z.B. Preemption nötig) zutrifft. Da die Größe des Bitvektors die Länge eines Maschinenwortes (üblicherweise mindestens 32 Bit) in der Regel nicht überschreitet, kann mit *einem einzigen* Test überprüft werden, ob mindestens ein Bit gesetzt ist. So entsteht insbesondere kein Mehraufwand für den durch den nebenläufigen Charakter der Sprache bedingten Test auf Preemption; er wird in einem Schritt gleich zusammen mit dem Test auf Überlauf der Halde, etc. mit erledigt, der für Sprachen mit Speicherbereinigung ohnehin nötig ist.

8.5 X-Register

In Emulatoren hat man den Vorteil, daß die Bereitstellung vieler allgemeiner (general purpose) Register sehr einfach und billig ist, während diese Ressource bei nativem Code sehr beschränkt ist. Da im Emulator die Register ohnehin im Hauptspeicher liegen kann man so problemlos einige Hundert (oder gar einige Tausend) Register allozieren und bezahlt dafür höchstens einige wenige Kilobyte an Speicher. Andererseits vereinfacht das Vorhandensein vieler Register den Compiler. Dieser kann davon ausgehen, daß unendlich viele X-Register vorhanden sind und bei Überschreiten der tatsächlich vorhandenen Anzahl die Compilation mit einer Fehlermeldung abbrechen (was in unserer Praxis bisher noch nie aufgetreten ist), die dem Benutzer anzeigt, daß ein internes Limit des Compilers erreicht wurde. Dadurch brauchen etwa Funktionen mit sehr vielen Argumenten nicht gesondert behandelt zu werden. Auch muß bei der Verwendung von temporären Variablen nicht unbedingt auf einen sparsamen Umgang mit X-Registern geachtet werden. Die schwierige Frage, welche der Variablen nun in X-Register kommen und welche in L-Registern abgelegt werden, stellt sich einfach nicht: man kann alle temporären Variablen auf X-Register abbilden.

In unserem Maschinenmodell besitzt nun aber jeder Thread seinen eigenen Satz an X-Registern. Das würde aber wieder bedeuten, daß man versuchen muß, die Anzahl der existierenden Register möglichst klein zu halten, um die Kosten sowohl in Laufzeit aber vor allem auch an Speicherplatz bei der Allokation eines Threads klein zu halten. Bei der Implementierung des Maschinenmodells sind wir daher einen anderen Weg gegangen. Der Emulator verwendet einen einzigen sehr großen X-Registersatz, der von allen Threads gemeinsam genutzt wird. Wenn der Scheduler die Kontrolle von einem Thread T_1 an einen anderen Thread T_2 abgibt, dann werden die von T_1 benötigten X-Register in T_1 gerettet (wie das im einzelnen geschieht werden wir im nächsten Abschnitt beschreiben). Umgekehrt wird vor Ausführung von T_2 der globale X-Registersatz mit den entsprechend in T_2 geretteten Werten initialisiert.

Es werden also nicht immer alle X-Register gesichert sondern nur diejenigen, die von einem Thread noch benötigt werden, die wir im folgenden als *lebendige* X-Register bezeichnen wollen. Wie wird nun aber bestimmt, welche X-Register lebendig sind? Grundsätzlich gibt es zwei Möglichkeiten, warum der Scheduler einem Thread T die Ausführungsressourcen entzieht: Preemption und Suspension. Bei Preemption ist die Bestimmung der lebendigen X-Register einfach, da Preemption nur an zwei Stellen, nämlich den Instruktionen `return` und `apply` erfolgen kann. Bei `return` enthält lediglich $\texttt{X}_0$ noch einen Wert (den Rückgabewert der Funktion) der zu retten ist. Ähnlich einfach ist die Situation bei

`apply` R n

da hier n die Anzahl der Argumente des Funktionsaufrufes angibt, die bereits in den ersten n X-Registern liegen. Das heißt, es müssen genau die ersten n X-Register gerettet werden.

Schwieriger ist die Bestimmung der lebendigen X-Register bei Suspension, da wesentlich mehr Instruktionen suspendieren können. Bei der Suspension etwa von `apply` kann wie bei Preemption verfahren werden. Wenn dagegen etwa eine Operation, wie zum Beispiel die Instruktion `plus` für die Addition suspendieren muß, weil etwa ein Argument noch ungebunden ist, reicht es nicht, lediglich die Argumente von `plus` zu retten. Da die Klassifikation von Variablen in temporär und permanent von der Lage von Funktionsaufrufen abhängt (vgl. Abschnitt 5.6.2), kann es sein, daß bei einer suspendierenden Instruktion die X-Register noch Werte enthalten, die von nachfolgenden Instruktionen benötigt werden.

Zur Bestimmung welche X-Register zu retten sind, existieren verschiedene Möglichkeiten:

(a) Jede Instruktion I, die suspendieren kann, wird mit einem zusätzlichen Argument lr versehen, in dem der Compiler Information darüber ablegt, welche Register bei Suspension gerettet werden müssen.

(b) Für jede Funktion f wird statisch die Zahl n bestimmt, die den maximalen X-Registerindex angibt, den der Code von f verwendet. Bei Suspension an beliebiger Stelle innerhalb von f werden dann immer die ersten n X-Register gerettet.

(c) Bei Suspension an einer Instruktion I werden dynamisch zur Laufzeit die auf I folgenden Instruktionen inspiziert und die Menge der zu rettenden Register bestimmt.

Die Variante (a) hat den Nachteil, daß die Codegröße deutlich zunimmt, da sehr viele Instruktionen prinzipiell suspendieren können, während in der Praxis dies nur für einen Bruchteil aller Instruktionen auch wirklich der Fall ist. Zudem kann auch die Codierung der Menge der zu rettenden Register recht aufwendig werden. Variante (b) hat diesen Nachteil zwar nicht, dafür werden hier aber in der Regel viel mehr Register gerettet, als tatsächlich nötig wäre. Die Version (c) schließlich erlaubt zwar eine exakte Bestimmung der lebendigen Register, ist aber sehr teuer, da sie mit einer abstrakten Interpretation der nachfolgenden Instruktionen verbunden ist.

Nach dem Explorieren verschiedener Alternativen haben wird uns bei der Implementierung von Mozart für eine Kombination aus (b) und (c) entschieden. Bei Suspension werden also zunächst immer die ersten n Register (abhängig von der zugehörigen Funktion f) gerettet. Das sind in der Regel zwar deutlich zu viele, in der Praxis macht es sich allerdings weder in der Laufzeit noch im Speicherverbrauch signifikant bemerkbar, ob man nun beispielsweise 10 statt 3 X-Register sichert. Unangenehme Folgen können allerdings bei einer späteren Speicherbereinigung auftreten, da dann auch die im Beispiel 7 überflüssig geretteten Register unnötigerweise besucht und kopiert werden, so daß mehr Datenstrukturen als nötig von der Speicherbereinigung gerettet würden. Daher setzen wir bei der Speicherbereinigung Variante (c) ein: hier wird für jeden suspendierten (nicht für die durch Preemption angehaltenen Threads) Thread die exakte Menge der vom obersten Auftrag benötigten X-Register bestimmt.

Ein Algorithmus, der das Vorgehen im Fall (c) detailliert beschreibt, wird in [Meh99] angegeben.

8.6 Besondere Aufträge

Die Arbeitsweise der Instruktion `return` haben wir so beschrieben, daß sie zunächst prüft, ob der ausführende Thread leer ist, und wenn nicht den nächsten Auftrag zur Ausführung bringt. Die Instruktion muß also stets einen Test durchführen, was ungünstig ist, da dieser Test ohnehin nur sehr selten erfüllt ist und die Instruktion `return` zudem sehr häufig (nach jedem Funktionsaufruf) ausgeführt wird.

Wir führen daher eine neue Instruktion `emptyStack` ohne Argumente ein. Bei Ausführung terminiert die Instruktion den ausführenden Thread und gibt die Kontrolle an den Scheduler zurück. Bei jeder Erzeugung eines neuen Threads wird nun auf dessen Stapel zuunterst ein Auftrag abgelegt, dessen Programmzähler auf diese Instruktion zeigt, während der L- und G-Zeiger des Auftrages nicht benötigt werden. Die Instruktion `return` kann nun davon ausgehen, daß sich stets noch ein Auftrag auf dem Thread befindet, und braucht dies daher nicht explizit zu testen.

Die gleiche Technik können wir auch zum Retten der X-Register (vgl. Abschnitt 8.5) verwenden: bei Suspension oder Preemption legen wir ganz oben auf dem Thread einen Auftrag ab, dessen PC auf die neue Instruktion `restoreXRegs` zeigt. Der L-Zeiger des Auftrages zeigt dann auf eine Kopie der geretteten X-Register, der G-Zeiger wird nicht benötigt. Die Ausführung der Instruktion `restoreXRegs` bewirkt dann, daß die geretteten Register in die globalen X-Register des Emulators kopiert werden, danach wird der nächste Auftrag vom Thread genommen und zur Ausführung gebracht. Somit braucht man für das Retten von X-Registern keine speziellen Datenstrukturen im Thread selbst vorzusehen und auch der Scheduler, der einen Thread zur Ausführung bringen will, muß sich nicht explizit um die Restauration kümmern.

Auf diese Weise lassen sich beliebige Spezialaufträge auf einem Thread ablegen, die ein großes Maß an Flexibilität erlauben, ohne daß dadurch die Ausführung der normalen Aufträge durch einen gesonderten Interpretierungsschritt in Mitleidenschaft gezogen würde. Wir haben bei der Implementierung von DFKI Oz und Mozart ausgiebig darauf zurückgegriffen: so gibt es hier außer den oben erwähnten Aufträgen, noch eine Reihe weiterer spezieller Aufträge, die unter anderem folgende Aufgaben übernehmen:

- Das Freigeben von aquirierten Locks.
- Retten und Restaurieren eines speziellen Registers (self), das zur Implementierung der objektorientierten Teile der Sprache benötigt wird.
- Zur Implementierung lokaler Berechnungsräume.
- Haltepunktverwaltung des Debuggers [Lor98].
- Profiling.
- Ausnahmebehandlung [Meh99].

8.7 Threads und Effizienz

Im folgenden wollen wir kurz aufzeigen, daß die Integration von Nebenläufigkeit in eine Programmiersprache auf dem Abstraktionsniveau von Oz nicht mit Nachteilen im Hinblick auf eine effiziente Ausführung von Programmen gekoppelt sein muß. Dies gilt vor allem dann, wenn man die nebenläufigen Konstrukte der Sprache gar nicht einsetzt (gracefully degrading), aber auch wenn man Threads intensiv verwendet muß dies nicht mit Performanzeinbrüchen verbunden sein.

So ist zunächst die Tatsache, daß jeder Thread seinen eigenen Stapel verwendet, unkritisch, da ja (in einer sequentiellen Implementierung) nur ein Thread zu einem Zeitpunkt rechnet. Es können also etwa die zur Verwaltung des Stapels nötigen Datenstrukturen beim Laden eines Threads in den Emulator genauso in Registern gehalten werden, wie bei der Implementierung sequentieller Programmiersprachen, die nur einen zentralen Stapel verwenden.

Mögliche Effizienzprobleme bleiben zum einen bei dem regelmäßigen Testen auf die Notwendigkeit der Preemption und zum anderen in dem Aufwand, der bei der Erzeugung von Threads und dem Scheduling betrieben wird. Zum ersten Punkt haben wir in Abschnitt 8.4 gesehen, daß hier keine Extrakosten entstehen, da dies mit der Signalbehandlung in einem Schritt gekoppelt werden kann.

Auch der Aufwand, der für das Scheduling selbst betrieben wird, stellt in der Praxis keinen relevanten Performanznachteil dar, auch wenn man ein wesentlich komplexeres Scheduling-Schema realisiert als das hier vorgestellte. Wichtig ist hier, daß die einzelnen Zeitscheiben lange genug gewählt werden, so daß die für die Umschaltung benötigte Zeit vernachlässigbar wird, wenn der einzelne Thread hinreichend lange rechnet. Bei modernen Multitasking-Betriebssystemen liegen die Zeitscheiben, die den einzelnen Prozessen zugewiesen, werden im Millisekunden-Bereich [Tan92], so daß es sinnvoll erscheint, auch die einzelnen Threads in Oz mit entsprechend langen Zeitscheiben zu versehen, auch wenn diese nicht auf Prozesse des Betriebssystems abgebildet werden. Gehen wir in einer Beispielrechnung von einer Zeitscheibe von 50ms aus, so bedeutet das, daß eine mit 200 MHz getaktete RISC Maschine (ein eher langsames Modell) in dieser Zeitspanne

$$(200 \times 10^6) \times (50 \times 10^{-3}) \;=\; 1 \times 10^7$$

(also zehn Millionen) Maschinenzyklen durchläuft. Dabei fallen dann einige hundert oder gar einige tausend Zyklen, die für das Scheduling verwandt werden nicht weiter ins Gewicht. Andererseits können innerhalb einer Sekunde etwa 20 Threads nacheinander bearbeitet werden, so daß für den Benutzer immer noch das subjektive Gefühl der parallelen Abarbeitung entsteht.

Die Darstellung von Threads in diesem Kapitel ist bewußt einfach gehalten. Erweiterungen in verschiedene Richtungen sind allerdings leicht integrierbar. So ist beispielsweise die Einführung von *Prioritäten* beim Scheduling einfach umzusetzen. Auch komplexere Strategien für das Scheduling der Threads, wie etwa Prioritäten, die an den Ressourcenverbrauch eines Threads gekoppelt sind, sind leicht realisierbar.

Auch die Erzeugung eines neuen Threads kann sehr schnell geschehen. In Oz muß dazu lediglich eine Datenstruktur mit einem Stapel alloziert werden. Zudem ist für Threads klar, wann sie nicht mehr referierbar sind, so daß sie über die Freispeicherliste verwaltet werden können. So benötigt auch die Erzeugung vieler kurzlebiger Threads nacheinder nur wenig Haldenspeicher. Da viele Threads recht kurzlebig sein können und daher dann in der Regel nur sehr wenige Aufträge enthalten, wird die initiale Größe des Stapels besonders klein gewählt. Erst wenn dieser Platz nicht mehr genügt, wird der Stapel beim ersten Vergrößern entsprechend deutlich expandiert. So benötigen Threads nur vergleichsweise wenig Platz (in Mozart weniger als 40 Worte), so daß die gleichzeitige Existenz von vielen Tausenden (suspendierten) Threads kein Problem darstellt. Erst wenn alle diese Threads auch gleichzeitig mit Rechenzeit bedient werden sollen, kann es sehr lange dauern, bis auch der letzte Thread an die Reihe kommt. Hierbei handelt es sich aber um kein prinzipielles Problem des nebenläufigen Programmierparadigmas. In einer sequentiellen Implementierung kann man nicht erwarten, daß man eine Aufgabe schneller erledigt, wenn man viele Dinge gleichzeitig in Angriff nimmt.

Obige Überlegungen werden von den Benchmarks in Abschnitt 10.7.8 bestätigt: die Erzeugung und Ausführung von Threads und die Kommunikation von Threads untereinander ist in Mozart extrem schnell und leichtgewichtig gelöst. Mozart liegt hier um Größenordnungen vor Java Implementierungen und nimmt eine Spitzenstellung unter vergleichbaren funktionalen Sprachen ein.

Zusammenfassung

- Threads werden so dargestellt, daß der Keller einfach vergrößert werden kann. Das erlaubt eine schnelle und Speicherplatz sparende Erzeugung von Threads.
- Drei Register der Maschine `PC`, `L` und `G` dienen zum *Cachen des obersten Auftrages* eines Threads.
- Der Test auf *Preemption* verursacht keine Mehrkosten: er kann effizient im Rahmen der *Signalbehandlung* durchgeführt werden, die mit einen einzigen Test auskommt. Man sieht hierzu jeweils ein Bit in einem speziellen Register für jedes Signal vor.
- *X-Register* werden in der Implementierung nicht pro Thread sondern einmal für die ganze Maschine repräsentiert. Bei Suspension und Preemption müssen daher die lebendigen X-Register gerettet werden. Das geschieht, indem für jede Funktion die maximal zu rettende Anzahl von X-Registern bei der Übersetzung bestimmt wird. Erst bei der Speicherbereinigung wird dann die exakte Menge der lebendigen X-Register bestimmt, indem die Instruktionen an denen angehalten wurde, entsprechend analysiert werden.
- Um bei der Ausführung eines neuen Auftrages den Test zu sparen, ob der Keller des Threads leer ist, wird zuunterst eines jeden Threads ein *spezieller Auftrag* abgelegt.
- Dieses Vorgehen wird erweitert und z.B. für die Implementierung von Locks, lokalen Berechnungsräumen, objektorientierte Programmierung, etc. verwandt.
- Die Zeitspanne, die jedem Thread zugeteilt wird, kann so groß gewählt werden, daß die Kosten für das *Scheduling* vernachlässigbar sind und für den Benutzer trotzdem der subjektive Eindruck der parallelen Ausführung von Threads entsteht.
- Threads in Mozart sind sehr leichtgewichtig. Mozart liegt um Größenordnungen vor Java und nimmt eine Spitzenstellung im Vergleich zu anderen nebenläufigen Sprachen ein.

Kapitel 9

Native Funktionen

Bisher gibt es in L nur die Möglichkeit, Funktionen zu verwenden, die auch in L selbst definiert wurden. In diesem Kapitel wollen wir aufzeigen, daß der Emulator sehr einfach dahingehend erweitert werden kann, daß aus L heraus auch Funktionen aufgerufen werden können, die in anderen Programmiersprachen (z.B. C, C++) implementiert wurden. Solche Funktionen wollen wir im folgenden als *native Funktionen* oder *Builtins* bezeichnen.

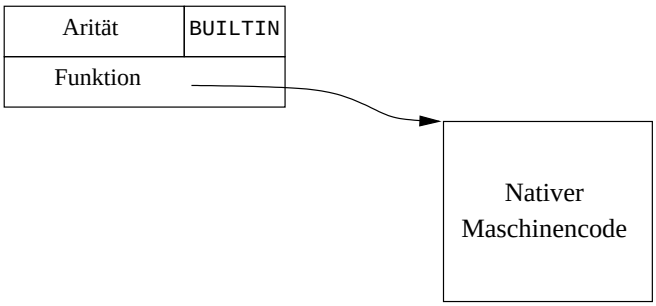
Für die Praxistauglichkeit eines Programmiersystems ist diese Form der Interoperabilität aus verschiedenen Gründen von großer Bedeutung. So ist beispielsweise der Zugriff einer Sprache auf Funktionalität des *Betriebssystems* (z.B. Funktionen zum Zugriff auf das Dateisystem oder zur Kommunikation mit anderen Prozessen) wichtig für die Akzeptanz durch den Benutzer. Darüber hinaus erlaubt die Einbindung von Builtins auch direkt den Zugriff auf bereits existierende *Bibliotheken* (z.B. zur Arithmetik, Stringverarbeitung, komplexe Datentypen, etc.), die in anderen Sprachen implementiert wurden, so daß diese nicht mehr in L selbst nachimplementiert werden müssen. Für einer Sprache, die unter Verwendung eines Emulators implementiert ist, kann es schließlich auch aus Gründen der *Effizienz* Sinn machen, bestimmte häufig verwandte Funktionen in einer hardwarenahen Sprache (wie etwa C) zu implementieren, auch wenn diese prinzipiell auch in L definiert werden könnten.

Wir werden in diesem Kapitel zeigen, daß es sehr einfach ist, in L Builtins verfügbar zu machen: im nächsten Abschnitt besprechen wir dazu zunächst deren Darstellung im Speicher. Anschließend zeigen wir in Abschnitt 9.2 wie die Applikation eines Builtins implementiert wird. In Abschnitt 9.4 beschrieben wird, wie bei der Suspension von Builtins vorzugehen ist. Abschließend gehen wir in Abschnitt 9.5 auf die Beziehung zwischen Builtins und den Operatoren von L ein.

9.1 Darstellung im Speicher

Native Funktionen werden vom Laufzeitsystem zur Verfügung gestellt. (In Mozart gibt es darüber hinaus auch die Möglichkeit für den Benutzer selbst Funktionen etwa in C/C++ zu implementieren und diese dynamisch zu laden [MMP+97]. Implementierungstechnisch werden diese genauso wie die vordefinierten Funktionen des Laufzeitsystems gehandhabt, so daß wir auf diesen Sonderfall im folgenden nicht näher eingehen müssen.) Aus Benutzersicht stellt sich die Situation dann so dar, daß bereits beim Systemstart eine Reihe von vordefinierten Bezeichnern existiert, die an Builtins gebunden sind.

Abbildung 9.1 Darstellung einer nativen Funktion im Speicher



Für den Benutzer soll die Verwendung von Builtins völlig transparent sein. Insbesondere können diese natürlich auch an beliebige Bezeichner gebunden werden und als Datenstrukturen frei weitergereicht werden. Wir repräsentieren Builtins im Speicher daher so wie in Abbildung 9.1 dargestellt. Analog zu benutzerdefinierten Funktionen (vgl. Abschnitt 7.3.5) enthalten Builtins zur Beschreibung ihres Typ eine sekundäre Marke `BUILTIN` und eine Zahl, die deren Arität angibt. Weiter wird dann nur noch ein Zeiger benötigt, der auf die interne Darstellung der Funktion, also den nativen Maschinencode, verweist.

9.2 Applikation

Da auch native Funktionen Datenstrukturen erster Klasse sind, kann man einer Funktionsapplikation der Form `f(x,y)` im allgemeinen nicht ansehen, ob es sich bei `f` um ein benutzerdefinierte oder eine native Funktion handelt. Native Funktionen benutzen daher den selben Aufrufmechanismus, wie benutzerdefinierte Funktionen: sie erwarten ihre Eingabewerte in Registern $\texttt{X}_0$ bis $\texttt{X}_{n-1}$, wenn n die Arität des Builtins ist, die Ausgabe liefern sie in $\texttt{X}_0$ zurück.

Die Definition der Instruktion `apply` R n muß dann entsprechend erweitert werden: falls R eine n-stellige native Funktion enthält, ruft der Emulator direkt die in der Struktur gespeicherte native Funktion auf (auch C++ unterstützt (ein bißchen) Funktionen höherer Ordnung). Da die Eingabeargumente zu diesem Zeitpunkt bereits in den X-Registern liegen, übergibt der Emulator einfach eine Referenz auf dieses Feld an das Builtin; insbesondere kann das Builtins dann auch direkt seinen Rückgabewert nach $\texttt{X}_0$ schreiben.

Der Aufruf eines Builtins terminiert nun allerdings nicht immer erfolgreich. So muß unter Umständen der Aufruf suspendieren (worauf wir in Abschnitt 9.4 näher eingehen); zudem kann ein Laufzeitfehler auftreten, wenn etwa die Argumente nicht vom korrekten Typ sind. Dies signalisiert das Builtin durch seinen Rückgabewert an den Emulator, der dann die geeigneten Maßnahmen ergreift. Ein Builtin ist somit eine C-Funktion, die ein Eingabeargument (ein Feld von Werten) erwartet und die ein Element des folgenden Aufzählungstyps als Ergebnis zurückliefert:

9.3 Determiniertheit

Bei einer Applikation der Form `f(x,y)` kann man analog wie bei benutzerdefinierten Funktionen Vorgehen (vgl. Abschnitt 4.7.2), wenn man statisch oder dynamisch erkennt, daß es sich bei `f` um ein Builtin handelt: man setzt dann statt der Instruktion `apply` eine spezielle Instruktion `applyBuiltin` ein, für die man sich das Dereferenzieren, den Typtest und den Aritätscheck sparen kann.

Auch im Hinblick auf die Registerallokation hat die Verwendung von `applyBuiltin` einen Vorteil: wenn man darauf achtet, daß Builtins höchstens das Ausgaberegister $\mathtt{X}_0$ verändern, kann man dies ausnutzen, indem man mehr Variablen als temporär klassifizieren kann (vgl. Abschnitt 5.6).

9.4 Suspension

Wenn der Aufruf an ein Builtin suspendieren muß, weil Eingabeparameter noch ungebunden sind, so signalisiert dies das Builtin, wie bereits erwähnt, an den Emulator indem es den Wert `SUSPEND` zurückliefert. Der Emulator muß dann den ausführenden Thread T suspendieren. Das Register `suspVars` dient nun dazu, dem Emulator diejenige Liste L von Variablen mitzuteilen, auf die T suspendieren muß; das Builtin legt also vor dem Rücksprung die Liste L in `suspVars` ab. Alternativ dazu könnte auch jedes Builtin selbst die entsprechenden Einträge in die Suspensionslisten der Variablen vornehmen, wodurch dies dann aber nicht mehr an zentraler Stelle im Emulator gebündelt wäre.

Wird ein Thread T, der bei der Ausführung eines Builtins suspendierte, wieder geweckt, weil eine der Variablen gebunden wurde, so wird das suspendierte Builtin noch einmal mit den gleichen Eingabeparametern aufgerufen; eine Suspension in der Mitte eines Builtins ist also nicht möglich.

9.5 Operationen und Builtins

Wenn man statisch erkannt hat, daß bei einer Applikation der Form `f(x,y)` die Variable `f` an ein Builtin gebunden ist, dann kann man auch alternative Aufrufkonventionen verwenden, indem man `x` und `y` nicht unbedingt in $\mathtt{X}_0$ und $\mathtt{X}_1$ übergibt, so daß der Compiler dann eine bessere Registerallokation vornehmen kann und man zudem eventuell Kopierbefehle zum Laden der Argumente in die X-Register spart. Man kann auch noch weiter optimieren und für bestimmte sehr häufig verwandte Builtins gleich eigene Maschineninstruktionen definieren, so daß man den in der Regel teuren Funktionsaufruf in C++ einspart. Dies entspricht dann gerade der Implementierung der Operationen von L (vgl. Abschnitt 3.8.1). Aus Implementierungssicht bedürfen die Operatoren von L dann keiner Sonderbehandlung mehr, da sie als Sonderfall von Builtins aufgefaßt werden können, die zudem den Vorteil haben, daß man sie als erster Klasse Datenstrukturen verwenden kann, was bei Operatoren nicht möglich ist.

Zusammenfassung

- Für die Praxistauglichkeit einer Programmiersprache ist der Zugriff auf Funktionen, die in anderen systemnahen Sprachen implementiert sind, von großer Relevanz. Wir nennen diese Funktionen *native Funktionen* oder *Builtins*.
- Im *Speicher* werden Builtins als Knoten mit einer direkten Referenz auf den nativen Maschinencode des Builtins dargestellt.
- Die Instruktionen für die *Funktionsapplikation* werden um einen zusätzlichen Test erweitert.
- Analog zur Applikation benutzerdefinierten Funktionen wird eine *Codespezialisierung* vorgenommen.
- Muß ein Builtin *suspendieren*, so teilt es über das Register `suspVars` dem Emulator mit, auf welchen Variablen zu suspendieren ist. Nach dem Wecken wird das Builtin noch einmal von vorne ausgeführt.
- Operationen von L sind ein Sonderfall von Builtins und können daher über Builtins implementiert werden.

Teil IV

Evaluierung

Kapitel 10

Evaluierung

In diesem Kapitel werden wir eine Evaluierung der Konzepte und Techniken vornehmen, die wir in den vorangehenden Kapiteln dieser Arbeit vorgestellt haben. Wir werden dabei so vorgehen, daß wir zum einen die Performanz von Mozart, die aktuelle Implementierung von Oz, mit anderen Systemen vergleichen werden. Zum anderen wollen wir an einer Reihe von größeren realen Anwendungen, die in Oz programmiert wurden, überprüfen, inwieweit sich die hier beschriebenen Techniken in der Praxis bewähren und wo gegebenenfalls noch Bedarf für Verbesserungen besteht.

Beim Vergleich von Programmiersystemen wird der Performanz der Ausführung von Programmen oft eine große Bedeutung beigemessen, obgleich für den Benutzer viele andere Faktoren (wie kurze Entwicklungszeiten, Portabilität, Quellcode- und Binärkompatibilität, Bibliotheken, Werkzeuge, Verfügbarkeit, leichte Erlernbarkeit) oft eine noch größere Rolle spielen. So wurde die Implementierung von Mozart auch nicht primär auf hohe Performanz um jeden Preis ausgelegt. Viele andere Kriterien spielten hier eine Rolle, die diesen Aspekt oft in den Hintergrund treten ließen. Wichtig für die Implementierung einer Sprache, die sich ständig stark im Wandel befand und noch befindet, war vor allem, flexibel auf neue Ideen eingehen zu können und diese in der Praxis leicht verifizieren zu können.

Nichtsdestotrotz können Benchmarks wichtige Hinweise sowohl für den Benutzer als auch den Implementeur geben. So erlauben sie dem Benutzer eine Einordnung unterschiedlicher Systeme. Da Anwendungen selten zu schnell laufen, kann der Benutzer zumindest grob abschätzen, mit welchen Einbußen oder Gewinnen er bei der Wahl des einen oder anderen Systems rechnen muß. Weiter können Benchmarks dem Benutzer zeigen, wie er Programme effizienter schreibt, indem sie ihm eine Vorstellung davon geben, welche Konstrukte eine Implementierung besonders gut unterstützt und welche weniger. Schließlich sind Benchmarks auch eine wichtige Hilfe für den Implementeur: sie zeigen diesem wo noch Bedarf für Verbesserungen besteht und können alle Komponenten wie Maschine, Compiler, Darstellung der Datenstrukturen betreffen. Bei Oz kommt weiter als Vorteil hinzu, daß das Design der Sprache noch nicht fix ist. So hatten nicht selten Ergebnisse von Benchmarks auch wieder Rückwirkungen auf das Sprachdesign, wenn man erkannte, daß sich bestimmte Konstrukte nicht oder nur mit großen Aufwand effizient implementieren lassen; oft reichte eine leichte Modifikation aus.

Im folgenden gehen wir in Abschnitt 10.1 kurz auf die Problematik der Durchführung eines fairen Vergleichs ein. In den Abschnitten 10.2 bis 10.5 stellen wir die Systeme vor, mit denen wir Mozart verglichen haben und beschreiben was und wie wir gemessen haben. In Abschnitt

10.7 präsentieren und diskutieren wir die Ergebnisse verschiedener Standard-Benchmarks. In Abschnitt 10.8 komplettieren wir das Material um empirische Daten aus größeren Anwendungen.

10.1 Das Problem eines fairen Vergleichs

Ein Ziel diese Kapitels ist eine Einordnung der Performanz von L relativ zu anderen vergleichbaren Programmiersprachen. Wir werden daher im folgenden Zahlenmaterial präsentieren, das diese Einordnung von L unter verschiedenen programmiersprachlichen Aspekten erlauben soll. Über das grundsätzliche Vorgehen und die Problematik zu aussagekräftigen Ergebnissen zu gelangen, wurde in der Literatur bereits ausgiebig diskutiert [Jai91]. Wir gehen daher im folgenden nur kurz auf diejenigen Faktoren ein, die in unserem konkreten Fall einen fairen Vergleich erschweren und wie wir deren Einfluß relativieren wollen.

L versus Mozart Ein Problem eines Vergleichs von L mit anderen Programmiersprachen besteht bereits darin, daß keine reale Implementierung von L existiert, da L ja nur als Vehikel für die Darstellung der Konzepte und Ideen dieser Arbeit dient. Da L bis auf den funktionalen Anteil aber eine echte Teilsprache von Oz ist, die wiederum der Implementierung Mozart zugrunde liegt, sind wir wie folgt vorgegangen: wir haben die Implementierung von Mozart erweitert, um die Rückgabe von Funktionsergebnissen à la L zu erlauben. Technisch ließ sich das am einfachsten durch die Einführung zweier neuer Builtins (zur Rückgabe des Funktionsergebisses und zum Laden des Ergebnisses in eine Variable) realisieren, die direkt auf zwei neuen Maschineninstruktionen übersetzt werden. Diese Erweiterung bedingte somit minimale Änderungen an Mozart: Definition zweier einfacher Instruktionen im Emulator und wenige Zeilen Extracode im Compiler.

Da diese Eingriffe mit nur möglichst wenig Aufwand verbunden sein sollten, ist der erzeugte Code allerdings nicht immer optimal: mitunter werden zum Beispiel noch zu viele `move`-Instruktionen erzeugt. Eine sorgfältigere Integration des funktionalen Teils von L in Mozart sollte also zum Teil noch etwas besser abschneiden, als es die im folgenden präsentierten Zahlen widerspiegeln.

In den folgenden Tabellen geben wir die Ergebnisse sowohl für L als auch für Mozart an, die zwar mit dem gleichen Emulator (und natürlich unter Verwendung der Oz Syntax) aber in unterschiedlicher Kodierung ermittelt wurden: dies erlaubt eine Abschätzung der Kosten einer rein funktionalen Implementierung gegenüber der Verwendung logischer Variablen.

Die Ermittlung der Werte für L unter Verwendung des Mozart-Emulators sind insofern nicht unfair (aus Sicht der anderen Systeme), als eine eigene Implementierung der schlankeren Sprache L mindestens genauso effizient sein muß, da diese nicht dem zusätzlichen Ballast der vollen Sprache Oz Rechnung tragen muß.

Äpfel und Birnen Wenn schon der Vergleich verschiedener Implementierungen der gleichen Programmiersprache schwer ist, so gilt das in noch viel stärkerem Maße für einen Vergleich völlig verschiedener Programmiersysteme. So wird man schon bei der Formulierung eines Algorithmus zur Lösung eines bestimmten Problems auf die Fähigkeiten der Programmiersprache, in der der Algorithmus implementiert werden soll, achten. Aber auch die Implementierung des gleichen Algorithmus kann in verschiedenen Sprachen deutlich unterschiedlich geschehen: soll man etwa einen Sortieralgorithmus mit Listen oder mit Feldern und destruktiven Änderungen in diesen Feldern implementieren? Was ist wenn

eine Sprache Felder und explizite Schleifen unterstützt, die andere aber nur Listen und rekursive Funktionen anbietet?

Um verläßliche Aussagen über die Performanz eines Programmiersystems machen zu können, sollte man sinnvollerweise so vorgehen, daß man verschiedene größere reale Anwendungen aus unterschiedlichen Bereichen auswählt (z.B. Arithmetik, symbolische Berechnung). Dann sollte man versuchen, diese möglichst effizient bei freier Wahl der Datenstrukturen und Algorithmen in jedem System zu kodieren. Das setzt allerdings nicht nur sehr gute Kenntnisse der Programmiersprache sondern auch der Eigenheiten der jeweiligen Implementierungen voraus. Eine solche Analyse können wir hier selbstverständlich nicht leisten. In dieser Arbeit geht es uns ja auch gar nicht um einen Vergleich der Qualität von Programmiersprachen. Vielmehr wollen wir hier klären, ob die Implementierung bestimmter programmiersprachlicher Konzepte wie Funktionen, Arithmetik, Listen, etc., in einer Multiparadigmen-Sprache wie Oz kompetitiv sein können.

Wir werden daher in Abschnitt 10.7 anhand verschiedener gängiger Benchmarks Vergleiche anstellen, die jeweils bestimmte Aspekte (z.B. Arithmetik, Funktionen, Erzeugung von Datenstrukturen, etc.) betonen. Dabei setzen wir jeweils die gleichen Algorithmen unter Verwendung gleichwertiger Datenstrukturen ein.

Speicherbereinigung Die Speicherbereinigung kann einen großen Einfluß auf die Gesamtlaufzeit haben. So können in Extremfällen die Zeiten für GC um ein vielfaches über den eigentlichen Laufzeiten der Benchmarks liegen, so daß man hier nur die Geschwindigkeit des Garbage Collectors und nicht die der eigentlichen Implementierung mißt. Viele Systeme erlauben eine Steuerung durch den Benutzer, wie häufig der Garbage Collector loslaufen soll. Was sind hier faire Einstellungen? Die Speicherbereinigung kann aber auch die Laufzeit der Benchmarks positiv beeinflussen: zum Beispiel durch besseres Cacheverhalten bei größerer Lokalität der Daten oder durch Verkürzung von Referenzketten speziell in Oz.

Wir haben daher versucht (soweit dies möglich war), bei den Testläufen den Garbage Collector jeweils auszuschalten oder haben die Zeiten für GC herausgerechnet.

Compiler Der Compiler hat einen entscheidenden Einfluß auf die Geschwindigkeit, mit der Programme ausgeführt werden. In diesem Kapitel geht es uns um die Untermauerung der These, daß eine kompetitive effiziente Implementierung einer Sprache wie L möglich ist. So messen unsere Testprogramme stets nicht nur die Geschwindigkeit der Implementierungen der virtuellen Maschinen sondern auch immer die Qualität der Compiler.

Zudem unterstützen viele Compiler auch noch eine Reihe von Optionen, mit denen man die Qualität des erzeugten Codes deutlich beeinflussen kann; dabei kann es vorkommen, daß ein Satz von Optionen, der für einen Benchmark von Vorteil ist, einen anderen benachteiligt. Wir haben daher alle Benchmarks mit den gleichen Compilerschaltern übersetzt, wobei wir (wenn vorhanden) einen Schalter verwandt haben, der generisch die höchste Optimierungsstufe aktiviert.

Kodierung Bei der Kodierung der Algorithmen muß darauf geachtet werden, daß man, auch wenn man gleiche Datenstrukturen und Anweisungen verwendet, im Detail jeweils bestmögliche Konstrukte einsetzt. Beispiel: mehrstellige Funktionen lassen sich in emuliertem OCAML am besten durch Currying formulieren, während die Verwendung von Tupeln deutlich langsamer ist (der native Code Compiler von OCAML hat dieses Problem nicht).

Architektur der virtuellen Maschine Wie wir in Abschnitt 5.1 bereits diskutiert haben, hat die Architektur der Maschine einen Einfluß auf die Effizienz der Implementierung. Vergleicht

man also etwa eine Register- mit einer Stackmaschine, so lassen sich daraus nur bedingt direkt Rückschlüsse auf die grundsätzlichen Möglichkeiten hinsichtlich der effizienten Implementierbarkeit der zugehörigen Sprachen schließen.

Nativ oder emuliert Der Vergleich eines nativen mit einem emulierten Systems ist per se nicht sehr aussagekräftig. Wir werden im folgenden aber dennoch auch Systeme aufnehmen, die nativen Code erzeugen, da dies interessante Einsichten erlaubt, welche Arten von Programmen wie stark von nativer Codeerzeugung profitieren. Diese Zahlen sind vor allem dann interessant, wenn man beide Implementierungstechniken anhand der gleichen zugrundeliegenden Programmiersprache vergleichen kann.

Hardware Die Wahl der Hardware ist von großer Bedeutung, sowohl für native als auch für emulierte Ausführung. So hängt davon etwa ab, wie natürlich (und damit wie effizient) sich der virtuelle Instruktionssatz auf den realen abbilden läßt. Auch die Unterscheidung ob RISC oder CISC Architektur ist wichtig: für virtuelle Maschinen mit vielen virtuellen Registern (wie die von L) ist es auf einer CISC Maschine schwerer, diese auf die dort in geringerer Zahl vorhandenen realen Register der Hardware abzubilden. Auch die Qualität der Cachearchitektur und (weniger) des Hauptspeichers der Hardware sind wichtig: Sprachen, die stark und zudem wenig lokal auf der Halde arbeiten, profitieren stärker von guten Caches, als Sprachen, die per Konstruktion mehr mit Registern oder lokal im Speicher arbeiten. In Abschnitt 10.3 gehen wir genau sowohl auf die Konfiguration als auch auf die Gründe für die Wahl des für die Benchmarks verwendeten Rechners ein.

Betriebssystem Nach unseren Erfahrungen hat die Wahl des Betriebsystems nur einen sehr untergeordneten Einfluß auf die Ausführungsgeschwindigkeit (insbesondere auf die relative Ausführungsgeschwindigkeit der verglichenen Systeme untereinander). So konnten wir etwa für Mozart bei Messungen unter Linux und unter Windows 95/NT praktisch keine Unterschiede feststellen, wenn beide Emulatoren mit der gleichen Version des GNU C Compilers übersetzt werden.

Last Die Last des Rechners kann die Ergebnisse beeinflussen. Bei den Messungen haben wir darauf geachtet, daß sonst keine anderen Jobs/Benutzer auf dem Rechner aktiv waren. Auf dem Rechner liefen insbesondere keine Serverprozesse (ftp, NFS, Mail, etc.).

Auswahl der Sprachen Bei der Auswahl der Sprachen und Systeme, mit denen wir Oz vergleichen wollen, haben wir unter der Unzahl der überhaupt existierenden Systeme eine Auswahl treffen müssen (siehe Abschnitt 10.2). Das betrifft sowohl die Sprachen selbst als auch konkreten Implementierungen einer bestimmten Sprache: ein Vergleich von Oz mit Perl macht genausowenig Sinn, wie ein Vergleich mit einer ineffizienten Implementierung von ML.

Installation Viele Systeme sind im Quellcode und/oder in Binärform verfügbar. Gerade wenn man einen Emulator selbst übersetzt, kann man leicht Fehler machen: übersetzt man nicht mit dem GNU C Compiler, so wird in der Regel kein threaded code (vgl. Abschnitt 6.6) verwandt. Aber auch wenn man den richtigen Compiler verwendet kann man durch eine Vielzahl von Schaltern für Optimierungen die Ausführungsgeschwindigkeit des Emulators beeinflussen. Wir haben daher darauf geachtet, wenn möglich die Binärversion für die Vergleiche heranzuziehen, da hier davon auszugehen ist, daß die Entwickler eine möglichst optimal übersetzte Version zur Verfügung stellen.

Tabelle 10.1 Verglichene Sprachen und deren Implementierungen

Sprache	Implementierung	Version	Nativ/ Emuliert	WWW
SML	SML of New Jersey	110	N	cm.bell-labs.com/ cm/cs/what/smlnj/
SML	Harlequin ML Works	1.0r2	N	www.harlequin.com/products
SML	Moscow ML	1.42	E	www.dina.kvl.dk/ ~sestoft/mosml.html
OCAML	Objective Caml emuliert	2.00	E	pauillac.inria.fr/ocaml
OCAML	Objective Caml native	2.00	N	
Java	Sun JDK	1.1.6	E	java.sun.com
Java	Kaffe	1.0.b1	N	www.kaffe.org
Prolog	Sicstus	3.7	E	www.sics.se/sicstus
Common Lisp	Allegro CL	5.0	N	www.franz.com/products
Common Lisp	GNU CL	2.2.2	N	
Erlang	Erlang System/OTP	4.6.4	E	www.erlang.se
C++	EGCS	1.1b	N	egcs.cygnus.com

Wir haben versucht, den Einfluß der oben genannten Faktoren bei unseren Messungen so weit als möglich zu reduzieren und größtmögliche Fairneß walten zu lassen. Dabei muß aber erwähnt werden, daß wir mit der eigenen Implementierung Mozart naturgemäß mit Abstand am vertrautesten sind. So ist der Einfluß von negativen Faktoren, die zu einem Verfälschen der Ergebnisse führen könnten, bei Mozart sicherlich am geringsten. Insbesondere deckten die ersten Ergebnisse in wenigen Benchmarks noch kleine Schwachstellen in Mozart auf, die wir dadurch beseitigen konnten (was ja auch ein Grund für Benchmarks ist). Die anderen Systeme hatten diese Chance nicht.

Das soll aber nicht heißen, daß wir kein Vertrauen in die Aussagekraft der ermittelten Werte haben: wir können uns nicht vorstellen, daß ein anderer Beobachter zu Ergebnissen kommen kann, die signifikant von unseren abweichen.

10.2 Sprachen und Systeme

Wir wollen im folgenden Mozart, die aktuelle Implementierung von Oz, mit der Implementierung verschiedener anderer Sprachen vergleichen. Tabelle 10.1 stellt alle verwendeten Systeme mit genauer Angabe der Versionsnummer und weiteren Verweisen ins Web zusammen. Bei der Auswahl haben wir darauf Wert gelegt, Sprachen zu berücksichtigen, die in ihrer Philosophie und Charakter mit dem von Oz möglichst einhergehen (C++ haben wir mit hinzugenommen, um einen Eindruck zum Verhältnis der Implementierungstechnologie imperativer Sprachen zu gewinnen). Diese Liste ist sicher nicht vollständig, berücksichtigt aber einen repräsentativen Ausschnitt moderner Sprachen dieser Art.

Bei der Auswahl der Implementierungen der einzelnen Sprachen haben wir besonderen Wert darauf gelegt, möglichst effiziente Vertreter auszuwählen. Diese sollten zudem über das Prototypen-Stadium hinaus sein und vielmehr ihre Praxistauglichkeit unter Beweis gestellt haben, indem sie für die Implementierung auch größerer Systeme geeignet sind.

Im folgenden vergleichen wir Oz unter Mozart mit den folgenden Programmiersprachen und deren Implementierungen:

ML *SML New Jersey* bildet den Quasi-Standard einer Referenz-Implementierung von SML. Es gilt allgemein als sehr effiziente Implementierung, die nativen Code für alle wichtigen Plattformen erzeugt. Die Distribution beinhaltet auch Concurrent ML [Rep92], eine Erweiterung von SML um Nebenläufigkeit.

MLWorks ist eine kommerzielle, native Implementierung von SML der Firma *Harlequin*.

Moscow ML ist ein beliebter Emulator für SML, der wegen seiner Kompaktheit und Portabilität geschätzt wird. Die Implementierung ist aus den Quellen von CAML (siehe unten) hervorgegangen.

Bei *OCAML* handelt es sich um eine Implementierung vom ML, die auf dem CAML Light Dialekt der Sprache beruht, die um ein klassenbasiertes Objekt-System erweitert wurde [RV97]. OCAML umfaßt sowohl einen Emulator als auch einen native Code Compiler. So können hier gut Vor- und Nachteile beider Techniken aufbauend auf der selben Basissprache verglichen werden.

Java *JDK* (Java Development Kit) ist die im UNIX Umfeld dominierende Implementierung des Sprachentwicklers Sun Microsystems. Entsprechend der aktuellen Bedeutung der Sprache, wird hier die Entwicklung im Vergleich zu den anderen Systemen mit entsprechend mehr Man-Power vorangetrieben. Wir haben die Blackdown Portierung für Linux verwandt, die die Original Quellen von Sun verwendet.

Kaffe ist ein JIT (just in time) natives System, dessen Quellen im Gegensatz zu JDK frei verfügbar sind.

Prolog *SICStus Prolog* vom SICS (Swedish Institute of Computer Science) dominiert des Feld der Prolog-Implementierungen (vor allem nachdem SICS die Rechte an Quintus Prolog erworben hat). Es wird gerne für Performanzvergleiche mit anderen logischen Sprachen herangezogen. Neben einem Emulator kann SICStus auch nativen Code für eine Reihe von Plattformen (leider nicht für die hier verwandten Intel-Prozessoren) erzeugen. Nach unseren Erfahrungen ist SICStus deutlich schneller als andere beliebte Prolog Implementierungen wie etwa SWI-Prolog oder Eclipse.

Common Lisp *Allegro Common Lisp* (ACL) von Franz Inc. ist einer der führenden Anbieter kommerzieller Common Lisp Implementierungen.

GNU Common Lisp (GCL) ist die freie GNU Implementierung der Sprache, die aus AKCL (Austin Kyoto CL) hervorgegangen ist. GCL wurde auch ausgewählt, weil es die Übersetzung von Lisp nach C als Implementierungstechnik verwendet.

Erlang wurde von Ericsson entwickelt [AVWW96] und erlebt zur Zeit einen großen Boom innerhalb der Firma. Es handelt sich um eine nebenläufige, funktionale, dynamisch getypte Sprache, die z.B. zur Implementierung von Telefonvermittlungsstellen (switches) eingesetzt wird.

Für die Tests haben wir *Erlang Systems/OTP* verwandt, das auf einem Emulator basiert und frei verfügbar ist. Daneben wird innerhalb von Ericsson noch an weiteren Implementierungen der Sprache gearbeitet, wie zum Beispiel Turbo-Erlang [Hau94], das nach C übersetzt.

C/C++ fällt aus dem Rahmen der anderen Sprachen heraus. Es wurde ausgewählt, um das Verhältnis zur Implementierungs-Technologie imperativer Sprachen abschätzen zu können.

Für die Messungen haben wir den EGCS Compiler verwandt, der die aktuellste Weiterentwicklung des GNU C Compilers darstellt.

10.3 Die Testplattform

Als Testplattform haben wir einen Pentium II basierten Arbeitsplatzrechner unter Linux gewählt, der im Einzelnen wie folgt ausgestattet ist:

CPU:	Intel Pentium II
Taktrate:	300 MHz
Anzahl CPUs:	2
Second-Level Cache:	512 kB pro CPU
Hauptspeicher:	512 MB, 60ns EDO
Festplatte:	4GB EIDE
Betriebssystem:	Linux 2.0.35 (RedHat 5.1)

Grund für die Wahl einer Intel basierten Plattform war (neben dem Kriterium der Verfügbarkeit) vor allem die Tatsache, daß es sich dabei um die mit Abstand am weitesten verbreitete Hardware-Plattform handelt. Somit haben die ermittelten Werte einen breiten Gültigkeitsbereich und es wird dem Leser erleichtert, die ermittelten Ergebnisse selbst nachzuvollziehen.

Der üppige Hauptspeicherausbau minimiert ein Verfälschen der Ergebnisse durch Paging und Swapping.

Wir bereits zuvor erwähnt hat die Wahl des Betriebssystems keinen signifikanten Einfluß auf die Meßergebnisse. Linux bietet den Vorteil, daß viele der verglichenen Systeme in Binärform verfügbar sind. So können diese einfach und in der von den Entwicklern gewünschten Form installiert werden. Zudem sind die meisten Systeme traditionell im UNIX-Umfeld entstanden und werden hier entwickelt, so daß diese Plattform in der Regel die beste Unterstützung findet.

10.4 So haben wir gemessen

Bei mehreren Messungen des gleichen Systems mit den gleichen Parametern können unter Umständen deutliche Schwankungen bei den Ergebnissen auftreten, die zum Beispiel durch Caching, Paging, etc. hervorgerufen werden. Diese gilt es möglichst zu reduzieren [Jai91]:

- Wir haben daher darauf geachtet, daß alle Benchmarks im physikalischen Hauptspeicher der Maschine ablaufen können, um den Einfluß des Pagings zu minimieren.
- Die Messungen wurden so vorgenommen, daß jeder einzelne Benchmark möglichst mindestens 1 CPU Sekunde benötigt. Die folgenden Tabellen enthalten für manche Systeme dennoch Zahlen, die deutlich darunter liegen: in diesen Fällen haben wir die zugehörigen Benchmarks entsprechend länger laufen lassen und die Ergebnisse herunter gerechnet.

- Für jeden Benchmark haben wir 10 Durchläufe vorgenommen und in die Tabellen das arithmetische Mittel dieser Ergebnisse aufgenommen.

Dadurch konnten wir erreichen, daß sich die Streuung der Werte in einem engen Rahmen bewegt: die Standard-Abweichung liegt nie mehr als 10 Prozent vom arithmetischen Mittel entfernt, meist sind es weniger als 5 Prozent Abweichung [Jai91].

10.5 Quellprogramme

Die Quellprogramme aller hier durchgeführten Benchmarks wurden über

```
http://www.ps.uni-sb.de/~scheidhr/thesis/benchmarks
```

verfügbar gemacht. Um dem Leser ein schnelleres Auffinden zu ermöglichen, haben wir zudem die Kodierung aller Funktionen in der Version für SML im Anhang C beigefügt.

10.6 Mikro Benchmarks

An dieser Stelle hatten wir ursprünglich vor, einen Vergleich der verschiedenen Systeme durchzuführen, der auf die Messung ganz bestimmter Aspekte fokussiert, wie zum Beispiel die Kosten für Funktionsaufrufe, das Erzeugen von Tupeln, Arithmetik, etc. Je mehr wir uns aber damit beschäftigten, umso schwieriger oder gar unmöglich erschien es uns, hierzu aussagekräftige Werte zu ermitteln, weswegen wir schließlich ganz darauf verzichten wollen. Die Gründe hierfür werden wir im folgenden darlegen.

Es gab sowohl Probleme darin, zu bestimmen, *was* zu messen ist als auch *wie* die Messungen zu erfolgen haben. Wir wollen dies am Beispiel der Arithmetik genauer erläutern; für die anderen Aspekte lagen die Probleme sehr ähnlich.

Will man die Kosten für einzelne arithmetische Operationen ermitteln, so gilt es zunächst zu klären, welche Operationen man messen will: so kann man zu sehr unterschiedlichen Ergebnissen kommen, wenn man eine Addition einer Multiplikation oder einem Vergleichstest gegenüberstellt (in Mozart werden beispielsweise manche Operationen durch eigene Maschineninstruktionen realisiert, andere durch native Funktionen). Weiter muß man entscheiden, ob man Ganzzahl- oder Fließkomma-Arithmetik oder auch Arithmetik auf großen Zahlen einbeziehen will. Unterschiedliche Ergebnisse ergeben sich in der Regel auch, wenn man eine Operation $x + y$, bei der beide Operanden zur Übersetzungszeit unbekannt sind, mit einer Addition etwa der Form $x + 5$ vergleicht (manchmal spielt auch der konkrete Wert der Konstanten noch eine Rolle). Will man hier nicht unfair sein, so kann das nur heißen, daß man Werte zu allen obigen Kombinationen ermittelt. So kann man leicht auf einige Dutzend Werte für jedes System kommen, was dem Leser dann sicher keinen Mehrwert mehr bringt.

Aber auch die Frage danach, *wie* die Messungen zu erfolgen haben, birgt Fallstricke. Wie soll man etwa die Kosten einer Addition $x + y$ ermitteln? Man könnte so vorgehen, daß man eine Schleife schreibt, in deren Rumpf man die Addition durchführt und die Kosten dafür mit einer leeren Schleife vergleicht. Mißt man dann aber wirklich nur die Kosten der Addition? Hier kommen mindestens die Kosten für das Bereitstellen der Argumente hinzu: wenn x und y in Registern

gehalten werden, ergeben sich völlig andere Werte, als wenn der entsprechende Compiler diese auf dem Keller ablegen muß. Unter Umständen sieht der Code für den Schleifenrumpf völlig anders aus. Zudem benötigt das Durchlaufen der Schleifen meist deutlich mehr Laufzeit als die Addition selbst. So sind Schwankungen der Ergebnisse von einer Messung zur nächsten oft mehr durch Schwankungen der Schleife bedingt als durch die Operationen, die man eigentlich messen will. Schließlich gilt es auch noch, die Cleverness des Compilers zu umgehen: man muß verhindern, daß dieser die Operation aus der Schleife herausfaktorisiert oder sie gar ganz eliminiert. Das kann dann aber leicht zu Konstruktionen führen, die wieder ganz andere Dinge messen als die, an denen man eigentlich interessiert ist.

Aus diesen Gründen haben wir auf eine Messung verzichtet und uns auf die Messungen in Abschnitt 10.7 konzentriert, die auch jeweils spezielle Aspekte betonen, aber unter Berücksichtigung der Bemerkungen aus Abschnitt 10.1 zu aussagekräftigeren Ergebnissen führen.

10.7 Standard-Benchmarks

Wir vergleichen im folgenden die unterschiedlichen Systeme anhand verschiedener in der Literatur häufig verwandter Programme zur Messung der Performanz der Implementierung von Programmiersprachen. Wir haben die Auswahl dabei so getroffen, so daß die verschiedenen Benchmarks jeweils unterschiedliche Implementierungsaspekte (wie z.B. Arithmetik, Funktionen, Erzeugung von Datenstrukturen, Matching, etc.) abdecken.

Wir haben nur die Laufzeit der verschiedenen Programme gemessen, den Speicherverbrauch haben wir nicht mit einbezogen: viele Systeme bieten dem Benutzer gar nicht die Möglichkeit diesen zu ermitteln. Bei anderen besteht der Speicher aus unterschiedlichen Segmenten, bei denen oft durch Verwendung einer Freispeicherverwaltung diese Werte schwer zu bewerten sind (so braucht in der Regel der erste Durchlauf mehr Speicher als die folgenden). Vor allem aber hat die Wahl der Parameter zur Steuerung der Speicherbereinigung einen entscheidenden Einfluß auf das Speicherplatzverhalten, so daß sich (wie wir in Abschnitt 10.1 bereits ausgeführt haben) ein fairer Vergleich nicht bewerkstelligen läßt.

Die Ergebnisse haben wir in Form von Balkendiagrammen dargestellt. Dabei stehen kleine Balken stets für gute Performanz. Die Balken sind jeweils mit absoluten Zahlen in Millisekunden und der relativen Performanz im Vergleich zu Mozart (L) beschriftet. Bei der relativen Performanz bedeutet ein negativer Exponent bessere Performanz im Vergleich zu Mozart. So bedeutet also etwa ein Wert von 1.5^{-1}, daß das entsprechende System genau anderthalb mal so schnell ist wie Mozart.

Zur leichteren Orientierung haben wir die Systeme in zwei Gruppen eingeordnet: der obere Teil der Balkendiagramme enthält jeweils die Systeme, die auf Emulatoren basieren, während die untere Hälfte die Systeme listet, die nativen Code erzeugen.

10.7.1 Takeushi

Bei der Takeushi-Funktion handelt es sich um einen klassischen Benchmark aus der Welt der funktionalen Programmierung [Gab85]. Der Benchmark mißt einfache arithmetische Operationen, aber vor allem Funktionsaufrufe, also Bereitstellen der Argumente, Unterprogrammsprung und Rücksprung. `tak(24,16,8)` führt zu rund 2,5 Millionen Funktionsaufrufen mit einem sehr

Tabelle 10.2 Funktionen: Berechnung von `tak(24,16,8)`. Links mit ganzen Zahlen, rechts unter Verwendung von CPS.

System	Integer	CPS
L	886 (1.00)	2613 (1.00)
Mozart	1283 (1.45)	2613 (1.00)
Sicstus	2574 (2.91)	—
Erlang	2106 (2.38)	10042 (3.84)
OCAML emul	790 (1.12^{-1})	2383 (1.10^{-1})
Moscow ML	1948 (2.20)	2912 (1.11)
JDK	1239 (1.40)	—
OCAML nativ	87 (10.18^{-1})	1026 (2.55^{-1})
SML/NJ	323 (2.74^{-1})	388 (6.73^{-1})
ML Works	240 (3.69^{-1})	1824 (1.43^{-1})
Allegro CL	233 (3.80^{-1})	—
GNU CL	1640 (1.85)	—
Kaffe	246 (3.60^{-1})	—
GNU C++	161 (5.50^{-1})	—

breiten Rekursionsbaum, der circa zu 75 Prozent aus Blättern besteht.

Tabelle 10.2 zeigt in der linken Spalte die gemittelten Ergebnisse zum Aufruf von `tak(24,16,8)`. Die ersten beiden Spalten, die mit „L“ und „Mozart“ beschrieben sind, zeigen die Unterschiede auf, die sich ergeben, wenn man Funktionen so implementiert, daß man die Erbebisse über Register oder respektive über logische Variablen an den Aufrufer zurückgibt.

Wir kommentieren die Ergebnisse stichpunktartig:

- Die Verwendung logischer Variablen kostet in Mozart gegenüber L einen Faktor von ca. 1,5 an Laufzeit.
- Unter den Emulatoren ist nur OCAML noch schneller als L.
- Mozart hält sich sehr gut: gleich auf mit Java aber noch deutlich besser als andere funktionale Sprachen (wie Moscow ML und Erlang), obgleich dies die Paradedisziplin der funktionalen Sprachen ist und Mozart wegen Verwendung logischer Variablen benachteiligt ist.
- OCAML ist ca 15 % besser als L und etwas weniger als doppelt so schnell wie Mozart.
- Unter den nativen Systemen fallen OCAML positiv und GNU CL negativ auf. Alle anderen liegen dicht bei einander.

- OCAML nativ ist sogar deutlich (Faktor 2) schneller als C++: OCAML verwendet einen deutlich kleineren Funktionsheader und benutzt Register zur Parameterübergabe. Bei anderer Wahl von Compiler-Schaltern für GNU C++ oder Verwendung von Funktionen mit mehr Argumenten können sich diese Resultate deutlich verschieben.

Die rechte Spalte der Tabelle 10.2 zeigt die Ergebnisse für eine Variante der Takeushi-Funktion, die wir der Benchmark-Suite von SML/NJ entnommen haben. Hier wird die Funktion direkt in der in [App92] beschriebenen Technik des Continuation Passing Style (CPS) kodiert (vgl. Abschnitt C.1.2). So mißt dieser Benchmark vor allem das dynamische Erzeugen von Funktionen und higher-order Funktionsaufrufe. Die Ergebnisse wieder in Stichpunkten:

- Für L und Mozart wurden dieselben Zahlen angegeben, da eine Kodierung durch Verzicht auf logische Variablen keine Vorteile gebracht hätte.
- In Prolog, Java und C++ war eine Kodierung nicht möglich, da die Sprachen die dynamische Erzeugung von Funktionen nicht unterstützen.
- Allegro CL und GNU CL versagen: es tritt ein Überlauf ein, da beide Lisp Varianten Endrekursion offenbar nicht bei higher-order Aufrufen optimieren.
- Erlang fällt stark zurück: higher-orderness scheint hier noch verbesserungsfähig.
- Die anderen Emulatoren liegen hier recht dicht beieinander und ML Works hat nur noch einen geringen Vorsprung vor diesen.
- Interessant ist auch ein Vergleich mit der Integer-Variante:
  - Es bestehen nur minimale Unterschiede zwischen Integer und CPS für SML/NJ: offenbar verwendet auch die aktuellste Implementierung noch die CPS-Technik.
  - OCAML nativ bricht deutlich ein: die CPS Variante ist ca. 12 mal langsamer und nur noch etwa 2,5 mal schneller als Mozart.

10.7.2 Fibonacci

Auch die Fibonacci Funktion gehört zu *den* Benchmarks aus der funktionalen Programmierung. Die Funktion ist ähnlich wie die Takeushi Funktion aus dem letzten Abschnitt gelagert. Fibonacci verwendet aber mehr Arithmetik, nur ein statt drei Argumenten und die Anzahl der Blätter im Rekursionbaum ist mit 50 Prozent deutlich geringer als bei Takeushi.

Tabelle 10.3 zeigt in der linken Spalte die Ergebnisse in der gängigen Variante unter Verwendung von Ganzzahl-Arithmetik. Im Vergleich zu Takeushi fällt Mozart deutlicher zurück; hier kommen verschiedene Faktoren zusammen: dynamische Typen wirken sich negativ bei der Arithmetik aus, ein registerbasiertes Maschinendesign (vgl. Abschnitt 5.1) und einfachere Verwaltung der L-Registersätze (vgl. Abschnitt 5.2) drücken auf die Ausführungsgeschwindigkeit von Funktionsaufrufen. Bei den anderen Systemen bietet sich in etwa das gleiche Bild wie auch bei Takeushi. Insbesondere liegen die nativen Systeme wieder ganz stark vorne.

Während sich in der Ganzzahl-Variante keine Überraschungen auftun, ergeben sich interessante Resultate, wenn man eine kleine Änderung vornimmt, indem man Fließkommazahlen statt ganzen Zahlen verwendet (rechte Spalte in Tabelle 10.3):

Tabelle 10.3 Funktionen und Arithmetik: `fib(31)`. Links mit ganzen Zahlen, rechts mit Fließkommazahlen.

System	Integer	Fließkomma
L	1332 (1.00)	1939 (1.00)
Mozart	1572 (1.18)	2215 (1.14)
Sicstus	3655 (2.74)	4688 (2.42)
Erlang	2258 (1.70)	6344 (3.27)
OCAML emul	767 (1.74^{-1})	2306 (1.19)
Moscow ML	1044 (1.28^{-1})	1868 (1.04^{-1})
JDK	1105 (1.21^{-1})	2068 (1.07)
OCAML nativ	102 (13.06^{-1})	869 (2.23^{-1})
SML/NJ	264 (5.05^{-1})	431 (4.50^{-1})
ML Works	181 (7.36^{-1})	1039 (1.87^{-1})
Allegro CL	231 (5.77^{-1})	3097 (1.60)
Allegro CL (decl.)	195 (6.83^{-1})	1866 (1.04^{-1})
GNU CL	3056 (2.29)	4223 (2.18)
Kaffe	224 (5.95^{-1})	1355 (1.43^{-1})
GNU C++	174 (7.66^{-1})	251 (7.73^{-1})

- Die Fließkommavariante ist bei OCAML emul und Erlang circa um den Faktor 3 langsamer als die Integer-Version, so daß OCAML nun etwa gleichauf mit Mozart ist.
- Mit Ausnahme von SML/NJ und C++ scheint dieser Benchmark allen nativen Systemen deutliche Probleme zu bereiten.
- Deutlich sind die Einbrüche bei ML Works und OCAML nativ: trotz strengen statischen Typsystems verwenden beide Systeme hier offenbar eine geboxte Darstellung für Fließkommazahlen und allozieren diese auf der Halde. Eine ungeboxte Darstellung von Fließkommazahlen wird wohl vorwiegend nur bei der Verwendung von Schleifen vorgenommen, nicht aber bei der Parameterübergabe [Ler97].
- Auch Kaffe hat hier Probleme und liegt nur noch wenig vor JDK.
- Am stärksten sind die Einbrüche bei ACL, das sogar hinter Mozart zurückfällt.
- Der Common Lisp Standard erlaubt auch explizit die Deklaration der Typen von Variablen. Fügt man solche Deklarationen hinzu, so werden die Ergebnisse für ACL besser (Zeile „Allegro CL (decl.)“). Der Compiler setzt dann aber keine Typtests mehr ein: ruft man die Funktion dann mit einem falsch getypten Argument auf, führt dies zu den un-

Tabelle 10.4 Listenoperationen: naives Umdrehen einer Liste mit 3000 Elementen.

Mozart	1101 (1.00)
L	1989 (1.81)
Sicstus	1752 (1.59)
Erlang	3970 (3.61)
OCAML emul	1663 (1.51)
Moscow ML	2876 (2.61)
JDK	13013 (11.82)
OCAML nativ	1143 (1.04)
SML/NJ	686 (1.60^{-1})
ML Works	1060 (1.04^{-1})
Allegro CL	2255 (2.05)
GNU CL	4877 (4.43)
Kaffe	27212 (24.72)
GNU C++	1873 (1.70)

terschiedlichsten Fehlern (von falschen Ergebnissen bis zu Abstürzen mit bus error). Der Leser mag selbst entscheiden, ob ein solcher Vergleich noch sinnvoll ist.

10.7.3 Naive Reverse

Bei diesem Benchmark handelt es sich um *den* Benchmark aus der logischen Programmierung [War77], bei dem es um das naive Umdrehen einer Liste geht, wobei die Funktion `append` (und damit das Erzeugen neuer Listenstrukturen auf der Halde) im Mittelpunkt steht. Hier sind Sprachen mit logischen Variablen im Vorteil, da sich `append` hier endrekursiv formulieren läßt. Die Zeile „Mozart“ in Tabelle 10.4 zeigt die Variante mit logischen Variablen, während die Zeile „L“ die übliche funktionale (und insbesondere nicht endrekursive) Definition verwendet.

- Durch die Verwendung von logischen Variablen läßt sich in Mozart eine Beschleunigung um den Faktor 1.7 erzielen.
- Mozart ist etwa gleich auf mit OCAML nativ und ML Works und doppelt so schnell wie natives ACL.
- Der Abstand der emulierten zu den nativen Systemen ist deutlich geringer als bei den bisherigen Benchmarks: bei Operationen auf der Halde können die nativen Systeme ihre Vorteile kaum noch ausspielen.
- Erlang enthält Zeiten für die Speicherbereinigung, wobei unklar ist, wie hoch diese sind.

Tabelle 10.5 Quicksort: 30-maliges Sortieren einer Liste mit 5000 Elementen.

System	Listen	Higher-Order	Felder
Mozart	1240 (1.00)	3619 (1.00)	3088 (1.00)
L	—	3189 (1.13^{-1})	—
Sicstus	3016 (2.43)	41772 (11.54)	—
Erlang	3763 (3.03)	7563 (2.09)	—
OCAML emul	1952 (1.57)	3079 (1.18^{-1})	3855 (1.25)
Moscow ML	2674 (2.16)	4318 (1.19)	11082 (3.59)
JDK	6437 (5.19)	6872 (1.90)	2085 (1.48^{-1})
OCAML nativ	478 (2.59^{-1})	656 (5.52^{-1})	770 (4.01^{-1})
SML/NJ	336 (3.69^{-1})	984 (3.68^{-1})	390 (7.92^{-1})
ML Works	384 (3.23^{-1})	686 (5.28^{-1})	280 (11.03^{-1})
Allegro CL	415 (2.99^{-1})	668 (5.42^{-1})	2031 (1.52^{-1})
GNU CL	3121 (2.52)	3092 (1.17^{-1})	23679 (7.67)
Kaffe	11912 (9.61)	12123 (3.35)	356 (8.67^{-1})
GNU C++	358 (3.46^{-1})	480 (7.54^{-1})	148 (20.86^{-1})

- Java fällt weit zurück (Listen wurden hier als Objekte implementiert): offensichtlich ist das Erzeugen neuer Objekte in Java sehr teuer.
- Emuliertes JDK ist etwa doppelt so schnell wie natives Kaffe.
- Mozart ist sogar deutlich schneller als C++:
  - Die Speicherverwaltung wurde nicht über `malloc`/`free` implementiert, sondern verwendet selbst geschriebene Routinen, die sehr viel effizienter sind, sonst wäre der Abstand noch deutlich größer.
  - Die Funktion `append` wurde funktional implementiert: durch Verwendung von Schleifen (was hier nicht ganz einfach ist) lassen sich bei C++ sicher noch deutliche Verbesserungen erzielen.

10.7.4 Quicksort

Wir betrachten 3 verschiedene Versionen des Quicksort-Algorithmus. Tabelle 10.5 zeigt in der linken Spalte eine Version, die mit Listen arbeitet. Die Version in der mittleren Spalte „Higher-Order“ unterscheidet sich davon, daß hier der Vergleichstest der einzelnen Argumente nicht direkt über den <-Operator sondern abstrakt über eine Funktion durchgeführt wird (die zur Laufzeit auch wieder an < gebunden ist), so daß die Compiler hier weniger gut optimieren können. Die

rechte Spalte „Felder“ schließlich zeigt eine Variante, bei der als Eingabe ein Feld verwandt wird, das beim Sortieren destruktiv geändert wird.

Als Eingabe diente jeweils eine Liste (bzw. ein Feld) mit 5000 Elementen, die durch einen Pseudo-Zufallszahlengenerator erzeugt wurden. Alle Systeme verwendeten hierzu die selbe Funktion, so daß die Eingaben jeweils gleich waren.

Der Vergleich logischer mit funktionalen Systemen wurde fair gestaltet. Die Partitionierung der Liste wurde so implementiert, daß nicht zwei Ausgaben (die sich unter Verwendung logischer Variablen effizienter realisieren lassen) erzeugt wurden, sondern die Funktion `partition` direkt den rekursiven Aufruf von `quicksort` startete (vgl. Abschnitt C.4.1).

- Der Benchmark mißt Listen-Operationen, Funktionsaufrufe mit vielen Argumenten und arithmetische Vergleiche.
- Für L ergeben sich nur in der higher-order Variante leichte Vorteile, weswegen wir die anderen Werte für L weggelassen haben.
- Listen
  - Mozart führt mit deutlichem Abstand die Emulatoren an: das liegt daran, daß Mozart in der inneren Schleife keinen L-Registersatz benötigt und damit alle Werte in Registern halten kann.
  - ACL schlägt sich gut.
  - SML/NJ ist sogar etwas besser als C++.
  - Bei Java wiederholt sich das Bild von naive reverse: Objekterzeugung ist teuer (vor allem in Kaffe).
- Higher-Order:
  - Alle Systeme verlangsamen mehr oder weniger stark.
  - Mozart braucht nun auch einen L-Registersatz und wird deutlich langsamer, liegt aber immer noch recht gut.
  - Higher-orderness in Prolog muß durch dynamische Erzeugung von Goals teuer erkauft werden.
  - Higher-orderness wurde in Java durch virtuelle Methoden simuliert: das verlangsamt den Benchmark aber nur wenig: offenbar dominieren die Kosten für das Erzeugen von Objekten.
  - In Erlang haben wir `apply(Module,Fun,[Arg1,Arg2])` statt einem echten higher-order Aufruf der Form `Fun(Arg1,Arg2)` verwandt. Letzteres führt zu einer Verlangsamung um den Faktor 2.
- Felder:
  - Mozart schneidet wider Erwarten recht gut ab: Felder wurden bei der Implementierung nicht besonders stark berücksichtigt (z.B. keine eigenen Maschineninstruktionen und sekundäre Marken, die die Typtests verlangsamen).
  - Mozart ist mehr als 3 mal schneller als Moscow ML.

Tabelle 10.6 Berechnung aller Lösungen des n-Damen Problems für $n = 10$.

Mozart	2101 (1.00)
Sicstus	5894 (2.81)
Erlang	5216 (2.48)
OCAML emul	2433 (1.16)
Moscow ML	2672 (1.27)
JDK	2318 (1.10)
OCAML nativ	158 (13.30^{-1})
SML/NJ	316 (6.65^{-1})
ML Works	374 (5.62^{-1})
Allegro CL	549 (3.83^{-1})
GNU CL	4037 (1.92)
Kaffe	1106 (1.90^{-1})
GNU C++	123 (17.08^{-1})

– Java ist in seiner Musterdisziplin (Felder und Arithmetik bei statischen Typen) nur mäßig vorne.
– Auch bei den nativen Systemen gibt es deutliche Unterschiede. Vor allem ACL liegt stark zurück (was sich durch Einfügen von Typdeklarationen sicher verbessern läßt).
– C++ markiert mit deutlichem Abstand den Stand der Kunst in dieser Disziplin.

10.7.5 N-Damen

Beim n-Damen Problem geht es darum auf einem $n \times n$ Schachbrett n Damen so zu plazieren, daß sie sich gegenseitig nicht schlagen können. Wir berechnen alle Lösungen für $n = 10$. Der Benchmark mißt vor allem Ganzzahl-Arithmetik (Vergleiche, Absolutbetrag, Addition, Subtraktion) und boolesche Verknüpfungen. Es wird nur mäßig auf der Halde gearbeitet, da die konstruierten Teillisten klein im Vergleich zum Suchraum sind.

Um einen fairen Vergleich zu erlauben, haben wir in C und Java Rekursion durch Schleifen ersetzt. In Java haben wir zudem die Funktion `abs` zur Berechnung des Absolutbetrages inline kodiert.

- Mozart führt die Emulatoren an und liegt sogar vor Java, wobei diese Applikation Java eigentlich besonders liegen sollte.
- Erwartungsgemäß liegen die nativen Systeme sehr weit vorne, da hier wenig auf der Halde gearbeitet wird.
- OCAML kommt schon sehr nahe an C++ heran.

Tabelle 10.7 Mandelbrot-Algorithmus

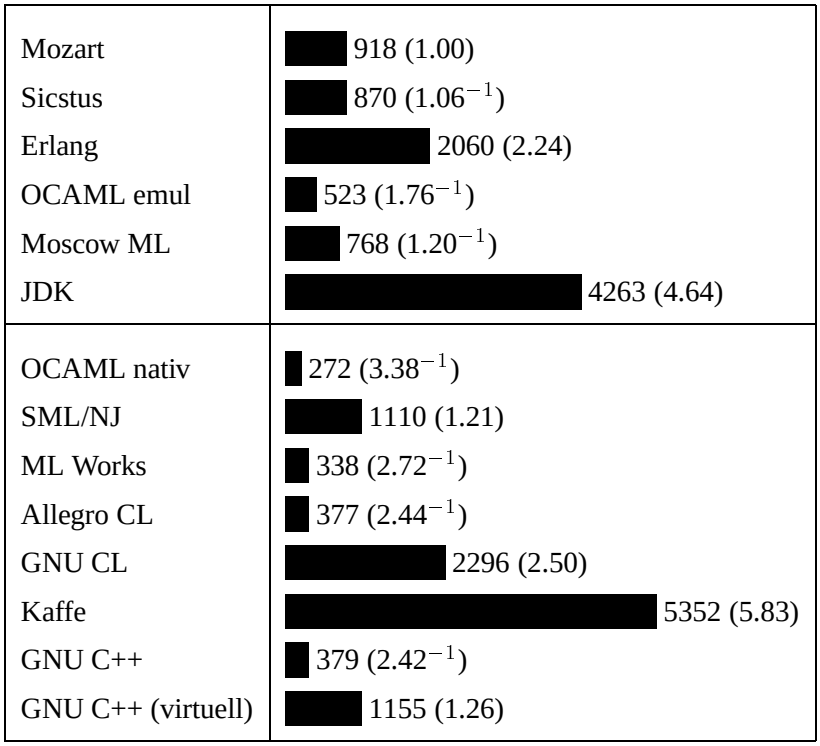
Mozart	3571 (1.00)
Sicstus	9468 (2.65)
Erlang	7502 (2.10)
OCAML emul	3211 (1.11^{-1})
Moscow ML	4711 (1.32)
JDK	2257 (1.58^{-1})
OCAML nativ	407 (8.77^{-1})
SML/NJ	318 (11.23^{-1})
ML Works	986 (3.62^{-1})
Allegro CL	4910 (1.37)
Allegro CL (decl.)	1540 (2.32^{-1})
Allegro CL (while)	240 (14.88^{-1})
GNU CL	7755 (2.17)
Kaffe	1350 (2.65^{-1})
GNU C++	91 (39.24^{-1})

10.7.6 Mandelbrot

Bei diesem Benchmark handelt es sich um einen Mandelbrot-Algorithmus aus den Benchmark-Suite für SML. Er mißt Schleifen in deren Inneren viele arithmetische (vorwiegend Fließkomma) Operationen vorkommen, die zudem auf die Werte globaler symbolischer Konstanten zurückgreifen.

- In Java und C wurden rekursive Funktionen durch geschachtelte Schleifen ersetzt, was eine deutliche Verbesserung brachte. In den ML Versionen brachte das keine Vorteile.
- Mozart ist in etwa gleich auf mit OCAML.
- JDK liegt unter den Emulatoren vorn: offenbar können Fließkommazahlen unboxed behandelt werden und müssen somit in Schleifen auch nicht auf der Halde verwaltet werden.
- Kaffe ist nur relativ wenig besser als JDK: vermutlich ist hier die Behandlung von Fließkommazahlen noch verbesserungswürdig.
- Eine direkte Umsetzung in ACL liefert ein mäßiges Ergebnis, das deutlich hinter Mozart liegt.
- Durch das Hinzufügen von Typdeklarationen in ACL (Zeile „Allegro CL (decl.)“) läßt sich eine Steigerung um mehr als den Faktor 3 erzielen, allerdings mit den zuvor bereits erwähnten Nebeneffekten (falsche Ergebnisse und Abstürze bei falsch getypten Eingaben).

Tabelle 10.8 Symbolische Ableitung: Pattern-Matching und Aufbau von Tupeln.



- Man kann ACL noch weiter beschleunigen, wenn man neben Typdeklarationen noch rekursive Funktionen durch Schleifen ersetzt (Spalte „Allegro CL (while)“): dann schlägt ACL sogar alle ML Implementierungen.
- C++ deklassiert die symbolischen Sprachen mit großem Abstand.

10.7.7 Symbolische Ableitung

Bei diesem Benchmark (in seiner ursprünglichen Form aus [War77] übernommen) geht es darum, 30 mal die 6. Ableitung nach x von

$$\left(\frac{1}{x}\right)^3$$

zu bestimmen. Dabei wird der obige Ausdruck und dessen Ableitungen durch ein Tupel dargestellt. Der Benchmark mißt Pattern-Matching und den Aufbau von neuen Tupeln auf der Halde.

Die dynamisch getypten Sprachen sind beim Pattern-Matching im Nachteil, da sie hierzu einen Hashing-Schritt durchführen müssen, wogegen bei den statisch getypten Sprachen ein einfacher Feldzugriff reicht.

In Java wurden die abzuleitenden Ausdrücke anders als in den anderen Sprachen nicht als Tupel sondern als Objekte verschiedener Klassen (je eine eigene Klasse zur Darstellung von Produkten, Konstanten, Variablen, etc.) dargestellt. Das Berechnen der Ableitung bestand nun in einem Aufruf der virtuellen Methode `deriv`, so daß die Fallunterscheidung hier implizit über virtuelle Methoden durchgeführt wurde. Analog wurde für C++ in der Zeile „GNU C++ (virtuell)“

vorgegangen, während bei „GNU C++“ eine explizite Fallunterscheidung durchgeführt wird.

Die Beobachtungen zu den Ergebnissen aus Tabelle 10.8:

- Mozart fällt deutlich hinter OCAML zurück, was vermutlich am teureren Pattern-Matching liegt.
- Zum ersten mal zieht Sicstus mit Mozart gleich.
- JDK fällt sehr weit hinter Mozart zurück, ist aber noch deutlich besser als Kaffe.
- SML/NJ bricht ein und ist sogar langsamer als Mozart. Unklar ist, ob das am Pattern-Matching oder dem Erzeugen von Tupeln liegt.
- OCAML ist deutlich schneller als C++.
- ACL liegt mit in der Spitzengruppe.
- Der Verzicht auf virtuelle Funktionen bringt in C++ einen Faktor 3.

10.7.8 Threads

In diesem Abschnitt vergleichen wir die Implementierung von Threads der verschiedenen Systeme. Keines der Systeme greift dabei auf die nativen Threads des Betriebssystems zurück, so daß die ermittelten Ergebnisse direkt miteinander vergleichbar sind.

Für beide Java-Implementierungen besteht unter Linux nicht die Möglichkeit der Verwendung von nativen Threads. JDK unter Solaris dagegen erlaubt die Benutzung grüner (= Threads, die in der Sprache selbst realisiert sind) und nativer Threads. Dort zeigt sich, daß Erzeugung und Scheduling von nativen Threads noch deutlich teurer als von grünen Threads ist. Man kann daher (bei aller Vorsicht) davon ausgehen, daß dies auch für Linux gelten würde, so daß die im folgenden präsentierten Ergebnisse für Java unter Linux mit nativen Threads noch deutlich schlechter ausfallen würden.

Tabelle 10.9 veranschaulicht die Kosten für die Erzeugung eines Threads: wir erzeugen 100.000 Threads mit leerem Rumpf. Die Zahlen in Tabelle 10.9 umfassen somit neben den Kosten für das eigentliche Erzeugen auch die Kosten für das Scheduling und die Freigabe der Threads nach deren Terminierung. Manche Systeme setzen die Ausführung zunächst mit dem neu erzeugten Thread fort, bevor der erzeugende Thread bearbeitet wird; andere Systeme verfahren umgekehrt. Um fair zu bleiben, wurde der Benchmark so implementiert, daß nach der Erzeugung eines neuen Threads die Kontrolle explizit zunächst an diesen übergeben wird, bevor mit der Erzeugung des nächsten Threads fortgefahren wird.

- Mozart liegt hier vorn, noch vor Concurrent ML.
- Thread-Erzeugung in Java ist extrem teuer, vor allem in Kaffe.
- Kaffe führt anscheinend keine Speicherbereinigung für Threads durch: nach der Erzeugung von ca. 10.000 Threads bricht die Ausführung mit der Meldung ab, daß der Speicher erschöpft ist. Die Ergebnisse wurden daher entsprechend hochgerechnet.

Tabelle 10.9 Erzeugung von 100.000 Threads

Mozart	349 (1.00)
Erlang	677 (1.94)
OCAML emul	910 (2.61)
JDK	17400 (49.86)
SML/NJ	404 (1.16)
Kaffe	92470 (264.96)

In einem weiteren Benchmark messen wir die Kosten für die Kommunikation und Synchronisation von Threads. Dazu verwenden wir ein einfaches Producer-Konsumer Szenario: ein Thread T erzeugt auf Anforderung eines zweiten Threads T' nach und nach alle natürlichen Zahlen; das heißt T' fordert eine Zahl an und suspendiert auf Antwort von T. T suspendiert dann wiederum so lange bis T' die nächste Zahl anfordert. Es werden also die Kosten für den Informationsaustausch (übermitteln einer Zahl) und die Kosten für das Suspendieren und Wecken von Threads gemessen.

In den einzelnen Sprachen haben wir den Kommunikationskanal unterschiedlich implementiert. In Oz haben wir einen Strom (eine am Ende offene Liste) verwandt: T' fordert eine neue Zahl an, indem er die Liste um eine Zelle, die eine logische Variable x enthält, verlängert. T liefert seine Antwort dann durch Unifikation mit x. In den ML Dialekten haben wir Kanäle zur Kommunikation verwandt [Nie97], in Erlang message passing mittels `!` (`send`) und `receive`. In Java verwenden wir shared memory zur Kommunikation und die Primitive `wait` und `notify` zur Synchronisation [Lea97].

Die ermittelten Ergebnisse sind somit mit Vorsicht zu genießen, da hier recht unterschiedliche Kommunikations- und Synchronisations-Mechanismen verwandt wurden. So bieten alle Sprachen eine Reihe von Alternativen, deren Verwendung zu deutlich anderen Ergebnissen führen könnte.

Tabelle 10.10 zeigt die Ergebnisse für 100.000 Kommunikationen, also das Anfordern der ersten 100.000 natürlichen Zahlen:

- Mozart, Erlang und SML liegen fast gleichauf.
- Die einzelnen Systeme liegen hier dichter beieinander als im vorangehenden Benchmark.
- Auffallend ist vor allem, daß OCAML um eine Größenordnung hinter Mozart zurückfällt.
- Java bricht nicht mehr so extrem ein, liegt aber immer noch ziemlich weit zurück.

Inspiriert von [WAMR97] zeigt Tabelle 10.11 als letztes Beispiel die Laufzeiten für die Berechnung von `fib(20)`, wenn man jeden rekursiven Aufruf der Fibonacci Funktion in einem eigenen Thread startet. Dieser Benchmark unterscheidet sich von den beiden vorhergehenden insofern, als daß hier nun jeweils sehr viele lauffähige Threads zu einem Zeitpunkt existieren. Zudem wird

Tabelle 10.10 Threads: 100.000 Kommunikationsoperationen

Mozart	628 (1.00)
Erlang	714 (1.14)
OCAML emul	7550 (12.02)
JDK	3129 (4.98)
SML/NJ	620 (1.01^{-1})
Kaffe	1600 (2.55)

bei jedem rekursiven Aufruf ein neuer Kommunikationskanal erzeugt; in Oz wurde hierzu eine logische Variable verwandt, in SML und OCAML wurde ein channel erzeugt. In Java braucht kein Kanal explizit erzeugt zu werden, hier wird das Ergebnis über das Thread-Objekt zurückgeliefert. Die Synchronisation erfolgt in Java über einen Aufruf von `join`.

- Kaffe bricht mit Speicherüberlauf ab.
- JDK bricht noch deutlicher ein als bei den vorangegangenen Benchmarks: offensichtlich kommt Java mit sehr vielen lauffähigen Threads und der damit verbundenen Kommunikation und Synchronisation besonders schlecht zurecht.
- Mozart und Erlang liegen gleichauf und wieder deutlich vor OCAML.
- SML ist noch schneller als Mozart, was vermutlich daran liegt, daß sich hier der Vorteil nativen Codes stärker bemerkbar macht.

10.8 Große Anwendungen

In den vorangegangenen Kapiteln dieser Arbeit haben wir schon an verschiedenen Stellen auf größere Oz-Anwendungen zurückgegriffen, um sowohl die statische Struktur als auch das dyna-

Tabelle 10.11 Threads: fib(20)

Mozart	144 (1.00)
Erlang	157 (1.09)
OCAML emul	681 (4.73)
JDK	101961 (708.06)
SML/NJ	102 (1.41^{-1})
Kaffe	—

mische Verhalten von realen Applikationen zu untersuchen. Diese Daten wollen wir in diesem Abschnitt vervollständigen, um die Effektivität der in dieser Arbeit vorgestellten Techniken in der Praxis zu beurteilen und um zu erkennen, wo eventuell noch Handlungsbedarf bei der Implementierung besteht.

Eine genauere Beschreibung aller Anwendungen, auf die wir im folgenden zurückgreifen, findet sich in Anhang B.

Alle Messungen eines bestimmten Aspektes wurden am selben Tag durchgeführt. Die Messungen verschiedener Aspekte können allerdings mehrere Tage und Wochen auseinander liegen. Da das gesamte System sich noch in der Entwicklung befindet (Emulator, Compiler, Bibliotheken und zum Teil auch die getesteten Anwendungen selbst) kann es dabei zu Differenzen bei gemeinsam verwandten Größen in verschiedenen Tabellen kommen. Nehmen also verschiedene Tabellen Bezug auf den Speicherverbrauch einer Applikation, so kann dieser für dieselbe Anwendung durchaus unterschiedlich beziffert werden. Dies hat allerdings keinen Einfluß auf die Aussagekraft der verschiedenen Messungen.

Alle diese Anwendungen wurden von erfahrenen Oz-Programmierern erstellt, die auch mehr oder weniger gut mit der Implementierung vertraut sind und somit ein gutes Performanzmodell der Implementierung der Sprache haben. Wir wollen daher zunächst kurz der Frage nach der Aussagekraft nachgehen, die die im folgenden untersuchten Werte haben.

Zunächst ist die Untersuchung „gut“ geschriebener Programme sicherlich sinnvoller als die solcher Programme, die naiv und ohne Beachtung gewisser Regeln erstellt wurden. So macht es beispielsweise sicher keinen Sinn, zu untersuchen, wie sich Programme verhalten, die von einem vorwiegend mit Pascal vertrauten Programmierer erstellt wurden, wenn dieser die Programme auch im Pascal typischen Stil in Oz implementierte.

Ein Problem der Programmiersprache Oz aus Sicht des Implementeurs ist einer ihrer großen Vorteile aus Sicht der Benutzer: die Sprache bietet eine Vielzahl von vordefinierten Datenstrukturen und Operationen auf diesen, die gepaart sind mit einer reichen Syntax, die viele Variationen zuläßt. Will man die Implementierung der Sprache wartbar und überschaubar halten, so ist es nicht sinnvoll und oft auch gar nicht möglich, alle diese Konstrukte gleich optimal zu unterstützen, zumal die gleichzeitige Optimierung verschiedener Aspekte sich oft gegenseitig ausschließt. Hier werden Kompromisse nötig. So muß man dem Programmierer wie in jeder anderen Sprache auch ein Performanzmodell an die Hand geben. Dies kann auch in Form einfacher Regeln erfolgen, die zeigen, welche Konstrukte welchen anderen vorzuziehen sind. Die Befolgung dieser Regeln bedeutet dann in der Praxis selten eine wirkliche Einschränkung: so können etwa bestimmte `case` Ausdrücke wesentlich besser übersetzt werden als andere; wenn nötig, ist ein einfaches Umformulieren fast immer möglich. Ähnlich leicht sind Regeln zu beachten, wie etwa die Verwendung statischer statt später Bindung bei Methodenaufrufen oder der Verzicht auf die geschachtelte Deklaration von Funktionen. Dabei gibt letzteres ein Beispiel für eine Aufgabe, die eigentlich oft auch vom Compiler durchgeführt werden könnte und sollte, während andere nur vom Programmierer selbst geleistet werden können.

Wenn man diese Regeln bereits schon beim Design einer Applikation beachtet, so zeigt sich in der Praxis, daß der Verzicht eines Konstruktes zugunsten eines anderen keine Einschränkungen beim Komfort des Programmierens bedeutet. Insbesondere gilt für Oz auch, was auf andere Programmiersprachen zutrifft: ein Großteil an Laufzeit und Speicher wird in einem sehr kleinen Teil des Programmcodes verbraucht. So sind wirkliche „Verrenkungen“ nur sehr selten nötig. So stand beim Design von vielen der im folgenden untersuchten Anwendungen der Punkt Effizienz nicht

Tabelle 10.12 Anteil verschiedener Datenstrukturen am Haldenspeicher in Prozent

Anwendung	Funktionen	Objekte	Records	Listen	Variablen	Summe
Prelude	4,8	1,8	18,3	37,5	11,0	73.4
Explorer	2,4	8,9	24,9	19,0	6,3	61.5
Browser	1,4	8,4	41,9	9,8	18,5	80.0
Gump (Scanner)	8,6	15,4	15,5	23,3	21,5	84.3
Gump (Parser)	8,5	10,4	15,8	33,8	21,5	90.0
Compiler	8,3	14,8	14,0	19,0	23,3	79.4
Scheduler (upper)	14,1	0,1	10,1	3,9	34,9	63.1
Scheduler (prove)	0,9	0,1	10,0	1,7	0,1	12.8
Spedition	2,4	0,8	21,8	44,6	9,3	78.9

im Mittelpunkt. Andere Kriterien, wie Kompaktheit, Übersichtlichkeit, Wartbarkeit. etc. spielten eine größere Rolle. So wurde oft erst nach Fertigstellung der Applikation (wenn überhaupt) nach Profilieren noch an wenigen Stellen nachgebessert.

10.8.1 Haldenspeicher

Tabelle 10.12 zeigt den Anteil ausgewählter Datenstrukturen am Haldenspeicher jeweils in Prozent zum gesammten verbrauchten Speicher. Die Summe der Zeilen ergibt jeweils weniger als 100 Prozent (siehe letzte Spalte), da neben den in Tabelle 10.12 dargestellten Werten, noch eine ganze Reihe weiterer Datenstrukturen von der Halde alloziert werden (z.B. Felder, Dictionaries, Threads, Suspensionen, etc.), die hier nicht berücksichtigt wurden. Insbesondere werden auch Threads, Suspensionslisten, Constraint-Variablen und lokale Berechnungsräume von der Halde alloziert, was sich vor allem in den Zahlen, für die Anwendungen aus dem Constraint-Bereich wie Scheduler und Explorer ausdrückt.

Die mit „Variable“ markierte Spalte zeigt nicht den Anteil von allen Variablen sondern nur den der Kellervariablen, auf die wir in Abschnitt 7.4.5 eingegangen sind. Es handelt sich hier also nur um diejenigen Variablen, die erzeugt werden müssen, weil in Oz eine Rückgabe von Funktionsergebnissen nicht über Register sondern über logische Variablen geschieht. Tabelle 10.12 zeigt den zum Teil sehr hohen Anteil dieser Variablen am Gesamtspeicherverbrauch und verdeutlicht, daß hier noch Handlungsbedarf und ein wichtiges Einsparpotential liegt.

Die Tabelle zeigt unter anderem den Anteil von Records (einer Verallgemeinerung der in dieser Arbeit besprochenen Tupel) an der Halde. Diese Zahlen enthalten nicht den Speicherverbrauch für Listen (wie in Abschnitt 4.3.4 beschrieben, werden diese in der Implementierung gesondert repräsentiert); Listen wurden in Tabelle 10.12 separat erfaßt. Diese Werte unterstreichen die Bedeutung der optimierten Darstellung von Listen: trotz Reduzierung des Speicherbedarf einer Liste auf die Hälfte gegenüber der nicht optimierten Darstellung liegt deren Speicherverbrauch bei den meisten Anwendungen noch über dem der Records und nimmt auch im Vergleich zu anderen Datenobjekten einen gehörigen Anteil am Gesamtspeicherverbrauch der Applikationen ein.

Die meisten Applikation wurden als objektorientierte Anwendungen angelegt, das heißt Objek-

Tabelle 10.13 Eigenschaften dynamisch erzeugter Funktionsabschlüsse.

Anwendung	Anzahl	Anteil am Heap	∅ Anzahl glob. Var.	Applikationen pro Abschluß
Prelude	6697	4,8%	1,4	23,1
Explorer	25660	2,4%	2,0	13,6
Browser	2362	1,4%	1,2	70,0
Gump (Scanner)	6136	8,6%	1,3	27,1
Gump (Parser)	143414	8,5%	1,1	61,3
Compiler	119037	8,3%	1,4	46,9
Scheduler (upper)	70672	14,1%	3,9	11,0
Scheduler (prove)	6360	0,9%	2,1	13,3
Spedition	1622	2,4%	2,8	64,5

te bilden die zentrale Datenstruktur. Die Implementierung von Objekten in Mozart ist nun so ausgelegt, daß ein Objekt mit n Attributen deutlich mehr Speicher benötigt als beispielsweise ein Tupel mit n Argumenten [Hen97]. Auffallend ist daher, daß sich dies nicht in den Zahlen in Tabelle 10.12 ausdrückt: hier liegt der Verbrauch an Speicher für Objekte durchgängig sehr moderat noch deutlich unter dem für Records und Listen. Das zeigt, daß ein radikalers Sprachdesign, bei dem alle Datenstrukturen als Objekte aufgefaßt werden, sicher auch signifikante Auswirkungen hinsichtlich des Speicherverbrauchs hätte. Es muß allerdings noch angemerkt werden, daß die Tabelle keine Aussage über die Lebensdauer der Datenstrukturen macht: bei Objekten ist es wahrscheinlicher, daß diese länger leben als etwa Listen. So verursachen Objekte trotz ihres geringen Anteils an der Halde unter Umständen mehr Aufwand bei der Speicherbereinigung (bei der momentanen Struktur des Mozart Garbage-Collectors, vgl. Abschnitt 7.1.3).

10.8.2 Erzeugung von Funktionen

Tabelle 10.13 zeigt Eigenschaften dynamisch erzeugter Funktionsabschlüsse. Die letzte Spalte der Tabelle zeigt, daß nach ca. 10 bis 70 Applikationen einer benutzerdefinierten Funktion ein neuer Abschluß erzeugt wird. Dennoch fällt deren Anteil am Gesamtspeicherverbrauch mit zum Teil deutlich unter 10 Prozent recht gering aus. So scheint hier kein Handlungsbedarf zur Integration fortgeschrittener Techniken, wie sie in Abschnitt 5.4.1 beschrieben wurden, zu bestehen.

Der Speicherverbrauch liegt sicher auch deshalb so niedrig, weil die durchschnittliche Anzahl von globalen Variablen mit selten mehr als 2 Variablen pro Abschluß sehr gering ausfällt. Hier zeigt sich die Bedeutung der in Abschnitt 4.7.2 beschriebenen Optimierung, die sich an dieser Stelle auszahlt: indem Referenzen auf globale Funktionen direkt in den Code geschrieben werden, brauchen diese nicht in die Abschlüsse mit aufgenommen zu werden.

10.8.3 Funktionsaufrufe

Tabelle 10.14 stellt die Verteilung von Funktionsaufrufen nach verschiedenen Kriterien dar. Alle Angeben in der Tabelle sind in Prozent. Wir beschreiben und diskutieren im folgenden die Bedeutung der einzelnen Spalten.

Tabelle 10.14 Verteilung von Funktionsaufrufen

Anwendung	opt. Aufrufe	Meth.-Aufrufe	opt. Builtins	nicht opt. Aufrufe	(davon Builtins)	HO-Aufrufe	Builtin Aufrufe
Prelude	48,0	3,5	25,1	23,2	(86,7)	4,2	45,2
Explorer	37,1	13,5	41,2	8,1	(67,5)	4,6	46,7
Browser	21,9	13,0	32,9	32,1	(59,9)	13,6	52,2
Gump (Scanner)	37,2	20,7	32,4	9,8	(37,0)	13,2	36,1
Gump (Parser)	47,3	3,1	45,4	4,0	(35,8)	5,0	46,9
Compiler	43,4	13,2	36,7	6,5	(28,4)	9,4	38,6
Scheduler (upper)	26,6	0,1	34,7	38,7	(48,6)	30,5	53,5
Scheduler (prove)	23,7	0,5	58,4	17,5	(62,1)	16,0	69,3
Spedition	37,2	0,8	42,3	19,6	(59,2)	12,4	53,9

Wir beginnen mit dem interessantesten Wert, nämlich der letzten Spalte „Builtin Aufrufe“. Diese zeigt den Anteil der Aufrufe an native Funktionen im Verhältnis zu allen Funktionsaufrufen. Es ergibt sich, daß mindestens jeder Dritte, oft sogar mehr als jeder zweite Funktionsaufruf einer nativen Funktion gilt. Dieser erstaunlich hohe Wert ist umso beachtlicher, da darin Aufrufe an solche Builtins nicht enthalten sind, die zu eigenen Maschineninstruktionen übersetzt werden (z.B. für bestimmte einfache arithmetische Operationen). Dies unterstreicht die Bedeutung von nativen Funktionen und die Wichtigkeit einer guten Behandlung des nativen Funktionsaufrufes. Die Zahlen geben zwar keinen Hinweis auf den Anteil der Laufzeit, die in Builtins verbracht wird, sie können aber eine Erklärung geben, warum beim Übergang auf native Implementierungstechnik bei praktischen Anwendungen oft kein allzu großer Performanzgewinn erzielt werden kann (vgl. Abschnitt 6.7).

Die Spalte „HO-Aufrufe“ zeigt den Anteil der higher-order Funktionsaufrufe, das heißt den Anteil an Funktionsaufrufen an benutzerdefinierte Funkionen, die nicht durch die in Abschnitt 4.7.2 beschriebenen Techniken optimiert werden können. Hier ergibt sich eine recht große Bandbreite, mit doch recht starken Spitzen in beide Richtungen. Der Mittelwert um 12 Prozent fällte aber doch recht gering aus, insbesondere wenn man berücksichtigt, daß die Angaben nur relativ zu den Aufrufen der benutzerdefinierten Funktionen nicht aber in Relation zu allen Funktionen sind (man beachte den zuvor festgestellten sehr hohen Anteil an Builtin-Aufrufen). So zeigt diese Zahl zweierlei: zum einen greifen die Techniken zur Optimierung der Applikation in einem sehr hohen Anteil aller Fälle und zum anderen werden Funktionen nur wenig higher-order benutzt.

Die restlichen Spalten zeigen die Verteilung von Funktionsaufrufen auf die verschiedenen Arten von Aufrufen. Die Spalte „opt. Aufrufe“ zeigt wieviele Funktionsaufrufe von benutzerdefinierten Funktionen durch die in Abschnitt 4.7.2 beschriebenen Techniken optimiert werden können (wohlgemerkt im Verhältnis zu allen Aufrufen also einschließlich Builtins). Die Spalte „Meth. Aufrufe“ gibt die relative Anzahl von Methodenaufrufen (Objektapplikation), die Spalte „opt. Builtins“ die von optimierbaren Aufrufen an Builtins an. Auch hier ergibt sich kein einheitliches Bild über die verschiedenen Anwendungen hinweg. Man kann aber erkennen, daß optimierte Aufrufe an Builtins und benutzerdefinierte Funktionen, sich in etwa die Waage halten. Demgegenüber werden meist nur wenige Nachrichten an Objekte unter Verwendung von später Bindung geschickt; bei Objekten hat der Programmierer in Oz auch die Möglichkeit, statische Bindung zu verwenden [Hen97], die in den Zahlen in der ersten Spalte der Tabelle enthalten sind.

Tabelle 10.15 Unifikation.

Anwendung	Var-Var	Var-Nonvar	Nonvar-Nonvar	(darunter Records)	opt. Unif.
Prelude	0,42	87,61	11,96	1,29	96,15
Explorer	4,04	86,94	9,03	0,10	81,29
Browser	0,36	88,69	10,95	0,94	88,24
Gump (Scanner)	4,34	81,78	13,88	0,59	93,35
Gump (Parser)	1,47	22,85	75,68	0,39	89,92
Compiler	3,24	48,81	47,95	0,29	92,58
Scheduler (upper)	0,61	93,30	6,10	0,29	57,34
Scheduler (prove)	27,93	49,45	22,62	0,01	16,34
Spedition	0,95	84,44	14,61	1,45	76,86

Die Spalten 4 und 5 zeigen, wieviele Aufrufe nicht statisch oder dynamisch optimiert werden können und somit durch die generische Instruktion `apply` aufgelöst werden. Auch hier ist die Verteilung recht inhomogen, die Werte liegen aber recht hoch. Grund hierfür ist die Tatsache, daß in Mozart bestimmte Aufrufe an Builtins nicht optimiert werden können. Wir haben daher in Klammern angegeben, wieviele nicht-optimierbare Aufrufe auf Builtins entfallen. In Mozart sind bestimmte Builtins als „situiert“ markiert, was dazu führt (wir wollen hier nicht weiter auf die Details eingehen), daß dafür momentan kein optimierter Aufruf erzeugt wird. Die Zahlen belegen aber, daß hier Handlungsbedarf besteht.

10.8.4 Unifikation

Tabelle 10.15 gibt Aufschluß über die Verwendung von Unifikation in den verschiedenen Anwendungen.Wieder sind alle Angaben in Prozent. Dabei haben wir dort nur die „echten“ Unifikationen berücksichtigt, das heißt viele Vergleiche über `case` wurden nicht berücksichtigt, wenn sie sich mit den in den Abschnitten 3.9 und 4.6 beschriebenen Techniken über einfache Vergleiche erschlagen ließen.

Die erste Spalte zeigt, bei wievielen Unifikationen beide Argumente logische Variablen sind. Besonders heraus fällt nur der Scheduler (prove), was allerdings nicht untypisch für eine constraintbasierte Anwendung ist, da in diesem Fall sehr häufig verschiedene Variablen über endlichen Bereichen miteinander unifiziert werden. Die Werte bei den anderen Anwendungen fallen zwar durchweg recht niedrig aus, können aber mit bis über vier Prozent doch einen nicht zu vernachlässigenden Anteil annehmen. Das kann zwei Gründe haben: zum einen kann es auf eine intensive Nutzung von Konstrukten wie zum Beispiel Differenzlisten (vgl. Abschnitt 5.3) deuten. Zum anderen kann hier aber auch ein Manko des Compilers die Ursache sein, indem zum Beispiel logische Variablen erzeugt werden, wo dies nicht zwingend nötig ist.

Die zweite Spalte der Tabelle zeigt, in wievielen Fällen es sich nur bei einer von beiden Argumenten einer Unifikation um eine logische Variable handelt. Dieser Fall entspricht in der Regel gerade der Situation, wo Funktionen ihre Rückgabewerte über logische Variablen an den Aufrufer kommunizieren. Man erkennt, daß dieser Fall den Löwenanteil einnimmt.

Die beiden nächsten Spalten zeigen den Anteil, bei denen beide Argumente keine Variablen sind,

wobei Spalte 4 angibt in wievielen Fällen es sich um Records handelt. Der Fall der Unifikation zweier Records ist mit selten mehr als einem Prozent praktisch unbedeutend; das zeigt, daß die Techniken zur Übersetzung des `case` gut greifen und fast alle Fälle erschlagen können. Bei allen anderen Fällen der Unifikation zweier Nicht-Variablen handelt es sich somit lediglich um einfache Tests auf Gleichheit, bei denen statisch keines der beiden Argumente bekannt war. Diese können durchaus einen beachtlichen Anteil annehmen. Besonders effizient behandelt der Emulator solche Tests momentan nur dann, wenn die Gleichheit positiv entschieden werden kann, andernfalls wird immer die Unifikationsprozedur aufgerufen. Im Falle von Ungleichheit kostet ein solch einfacher Test daher deutlich mehr: hier bestehen somit noch Verbesserungsmöglichkeiten durch Erweiterung der Tests direkt innerhalb des Emulators oder durch Einführung zusätzlicher spezieller Instruktionen.

Die letzte Spalte zeigt, wie oft eine optimierte Unifikation durchgeführt werden kann: in vielen Fällen wird die eigentliche Unifikationsroutine nämlich gar nicht erst angesprungen, da viele einfache Fälle bereits direkt von der entsprechenden Instruktion innerhalb des Emulators behandelt werden können. Die Unifikationsroutine wird dann in der Regel nur bei der Unifikation zweier Variablen oder einem Test auf Gleichheit angesprungen, wenn dieser nicht positiv entschieden werden kann. Mit Ausnahme des Schedulers als Constraint-Anwendung ergibt sich hier zwar eine Abdeckung von um 90 Prozent, diese könnte aber wie zuvor besprochen bei Tests auf Gleichheit noch verbessert werden.

10.8.5 Threads

Abbildung 10.16 veranschaulicht, wieviele Threads zur Laufzeit erzeugt werden und relativiert diese Zahl, indem sie diese ins Verhältnis zur Anzahl der Applikation benutzerdefinierter Funktionen setzt. Hier reicht das Spektrum von Anwendungen, die vollständig sequentiell ablaufen (wie dem Compiler), bis zu solchen, die extrem viele neue Threads absetzen. So erzeugt beispielweise der Explorer im Durchschnitt nach fast jedem vierzehnten Aufruf einer benutzerdefinierten Funktion einen neuen Thread; beim Scheduler ist dies gar nach jedem vierten Funktionsaufruf der Fall. Auch die meisten anderen Anwendungen machen durch die Erzeugung von zum Teil einigen hundert Threads regen Gebrauch vom nebenläufigen Charakter der Sprache. Diese Zahlen unterstreichen die Wichtigkeit einer effizienten und Ressourcen schonenden Implementierung von Threads in Mozart.

Zusammenfassung

Wir fassen die wichtigsten Ergebnisse dieses Kapitels wieder stichpunktartig zusammen.

Wir beginnen mit den Ergebnissen der Benchmarks aus Abschnitt 10.7:

- Mozart nimmt unter den Emulatoren eine Spitzenposition ein (jeweils mit einzelnen positiven und negativen Ausreißern): es ist etwa gleich auf mit OCAML und Java, in der Regel deutlich schneller als Moscow ML und etwa doppelt so schnell wie Sicstus Prolog und Erlang.

Tabelle 10.16 Anzahl der von verschiedenen Applikationen erzeugten Threads.

Anwendung	Anzahl Threads	Applikationen/ Threads
Prelude	87	1743,8
Explorer	25.139	13,6
Browser	259	633,2
Gump (Scanner)	5	33.577,0
Gump (Parser)	7	1.257.482,0
Compile Browser	1	5.594.734,0
Scheduler (upper)	2.910	268,0
Scheduler (prove)	20.877	4,1
Spedition	326	321,2

- Die nativen Systeme spielen ihre Vorteile dann aus, wenn wenig oder nur lokal im Speicher gerechnet wird: bei Arithmetik ist beispielsweise leicht eine Differenz von einer Größenordnung möglich.
- Werden viele Strukturen auf der Halde erzeugt, fällt der Abstand weitaus geringer aus (etwa Faktor 2 bis 3). Es kann auch vorkommen, daß Mozart in einzelnen Fällen schneller als ein natives ML System oder auch als C++ ist.
- Threads in Mozart sind sehr leichtgewichtig: sowohl in den Kosten für die Erzeugung eines Threads als auch in Kosten für die Kommunikation. Java fällt hier extrem zurück, obwohl es sich bei der Nebenläufigkeit um eines der zentralen Konzepte der Sprache handelt. Besonders die Kosten für die Erzeugung von Threads liegen hier um fast 2 Größenordnungen hinter Mozart.
- GNU CL ist in fast allen Benchmarks (zum Teil sehr viel) langsamer als Mozart: dies zeigt, daß die Übersetzung nach C per se noch keine Garantie für hohe Performanz darstellt. Auch hier muß viel Energie und Feinarbeit in den Compiler investiert werden, wenn man zu brauchbaren Ergebnissen kommen will.

Hier nun die wichtigsten Erkenntnisse aus Abschnitt 10.8:

- Der Anteil von Variablen am Haldenspeicher ist recht hoch: Rückgabe von Funktionsergebnissen über logische Variablen macht sich hier deutlich bemerkbar, auch wenn der Speicher durch GC wiederverwandt werden kann.
- Viele Records sind Listen, was eine optimierte Darstellung von Listen rechtfertigt.
- Der Anteil der Objekte an der Halde fällt unerwartet niedrig aus, obgleich fast alle Anwendungen Objekte als zentrale Datenstrukturen verwenden.
- Funktionsabschlüsse benötigen nur sehr wenig Speicher. Die durchschnittliche Größe eines Abschlusses fällt mit weniger als 2 Variablen sehr niedrig aus. Hier zeigt sich die Effektivität der Codespezialisierung von Funktionsaufrufen.

- Fast die Hälfte aller Funktionsaufrufe entfällt auf native Funktionen.
- Higher-order Aufrufe liegen relativ niedrig zwischen 5 und 15 Prozent (bei einer Applikation 30 Prozent) von allen Aufrufen an benutzerdefinierte Funktionen.
- Ein Großteil aller Aufrufe können durch Spezialisierung optimiert werden. Viele Aufrufe, die nicht optimiert werden, gehen an bestimmte (sog. situierte) native Funktionen. Hier besteht Handlungsbedarf.
- Nur wenige Funktionsaufrufe gehen an Methodenaufrufe unter Verwendung später Bindung; es gehen deutlich mehr Aufrufe an benutzerdefinierte Funktionen und statisch gebundene Methodenaufrufe.
- Wirkliche Unifikationen sind in der Praxis sehr selten. Meist geht es hier nur um das Binden von Variablen oder einfache Gleichheitstests von skalaren Werten. Die Behandlung letzterer kann in Mozart noch verbessert werden: wenn die Gleichheit nicht sofort entschieden werden kann, wird die eigentliche Unifikationsroutine aufgerufen.
- Threads werden unterschiedlich stark eingesetzt, fast alle Anwendungen erzeugen jedoch recht viele Threads. Das gilt sowohl in absoluten Zahlen (z.B. bis 25.000 Threads) als auch relativ (je ein neuer Threads nach 4 Funktionsaufrufen). Das unterstreicht die Akzeptanz und die Wichtigkeit einer effizienten und Ressourcen schonenden Implementierung von Threads.

Kapitel 11

Verwandte Arbeiten

In diesem Kapitel wollen wir auf verwandte Arbeiten eingehen, die bisher noch keine ausreichende Erwähnung fanden. Wir konzentrieren uns auf Unterschiede und Gemeinsamkeiten aus Implementierungssicht und gehen auf Aspekte des Sprachdesigns nur ein, sofern diese für die Implementierung in Zusammenhang mit den Themen dieser Arbeit von Bedeutung sind.

11.1 Multilisp und Futures

Multilisp [Hal85], eine parallele Scheme Implementierung, benutzt Futures als zentrales Konstrukt zur Einführung vom Nebenläufigkeit in die Sprache. Hier kann man eine Variable wie folgt deklarieren (unter Verwendung der L Syntax):

let val x = future(e) **in** e' **end**

Ein Thread T, der einen solchen Ausdruck auswertet, startet hierzu einen neuen Thread T', der e auswertet, während T mit der Berechnung von e' fortfährt. Wenn nun der Wert von x benötigt wird, aber noch nicht berechnet ist, wird die Auswertung von T gestoppt, bis T' terminiert. Multilisp schreibt nicht zwingend vor, daß die Auswertung von e in einem neuen Thread zu erfolgen hat und verlangt auch keine Fairneß beim Scheduling.

Die Verwendung von Futures in Multilisp im Zusammenhang mit Nebenläufigkeit ähnelt nun sehr der von logischen Variablen in Oz (wenn auch bei Multilisp der Fokus auf einer parallelen Implementierung lag): sie dienen beide zur Kommunikation und Synchronisation von Threads. Futures haben aber einen wesentlichen Vorteil gegenüber logischen Variablen: bei Futures ist klar festgelegt, welcher Thread die Variable binden darf. Bei logischen Variablen gilt das nicht; hier kann prinzipiell jeder Thread eine logische Variable durch Unifikation binden und dadurch die Berechnung korrumpieren.

Aus diesem Grund wurden Futures in Mozart aufgenommen: durch den Aufruf einer Basis-Operation läßt sich zu einer logischen Variable x eine Future x_f erzeugen, die quasi wie eine schützende Hülle um x gelegt wird. x_f kann man nun beliebig weiterreichen; ein lesender Zugriff oder auch der Versuch, x_f zu binden, führt zur Suspension, was implementierungstechnisch wieder am einfachsten zu lösen ist, wenn man auch Futures als spezielle Form von Variablen darstellt (vgl. z.B. auch Abschnitt 5.7). Nur solche Threads, an die man x direkt weitergibt, können x auch binden. Die aktuelle Implementierung von Futures in Mozart hat einen noch eher

experimentellen Status: implementiert man `append` unter Verwendung von Futures an Stelle von logischen Variablen, büßt man etwa eine Größenordnung an Laufzeit und fast ebensoviel an Speicher ein. Mögliche Optimierungen sind bereits angedacht, müssen aber noch umgesetzt und erprobt werden.

Während Multilisp nie wirklich effizient implementiert wurde, beschreiben Kranz et al. [DAK89] eine effiziente Implementierung von Mul-T, die etwa 100 mal schneller ist als Multilisp und insbesondere auch Futures unterstützt. Allerdings muß hier für eine Future eine sehr große Datenstruktur (vergleichbar einem Thread) alloziert werden, die in Mul-T auch nicht über eine Freispeicherliste verwaltet werden kann. Demgegenüber benötigt in Mozart eine logische Variable x nur dann mehr Speicher, wenn auf x suspendiert wird, bevor die Berechnung des Wertes von x terminiert hat (vgl. Abschnitt 7.4.6). Der Speicher eines Threads kann nach dessen Terminierung wiederverwandt werden, ebenso der Speicher einer Variablen nach deren Binden.

Mul-T bezahlt einen deutlichen Preis für die Bereitstellung von Futures, auch wenn diese nicht benutzt werden: gegenüber dem sequentiellen Basissystem T3, aus dem das parallele Mul-T hervorging, benötigt ein Benchmark, der ohne Futures auskommt, in Mul-T fast doppelt soviel Laufzeit [DAK89]. Die Performanz von Mul-T wird nicht in Relation zu anderen Lisp-Systemen gesetzt.

11.2 Concurrent ML

Reppy beschreibt mit Concurrent ML (CML) eine nebenläufige Erweiterung von SML/NJ [PR97, Rep91, Rep92]. Die Erweiterung kommt (neben einer kleinen Änderung am Compiler) ganz ohne Eingriffe in der Implementierung von SML aus. Somit ist CML komplett in SML implementiert.

Die Implementierung basiert im Wesentlichen auf zwei bereits vorhandenen Schnittstellen von SML/NJ: zum einen werden Threads durch die Verwendung von first-class Continuations (`callcc`) implementiert. Zum anderen wird für preemptives Scheduling die Möglichkeit der Behandlung asynchroner Signale in SML verwandt.

CML bietet als Basiskonstrukte die Möglichkeit der synchronen Kommunikation über Kanäle. Darauf aufbauend stehen sogenannte Events als first-class Werte zur synchronen Kommunikation zur Verfügung.

CML profitiert von der Tatsache, daß die Implementierung von SML/NJ auf der CPS-Technik [App92] basiert. Deshalb kommt SML ohne die Verwendung eines Kellers aus. Das wiederum bedeutet, daß die Erzeugung eines Threads in CML sowohl in der Laufzeit als auch im Speicherverbrauch sehr billig ist, da auch hier für die einzelnen Threads keine Keller erzeugt werden müssen, es reicht die Erzeugung einer vergleichsweise kleinen Datenstruktur auf der Halde.

Neben CML gibt es noch eine Reihe weiterer nebenläufiger Erweiterungen von ML [Nie97]. Poly/ML [Mat97], eine kommerzielle SML Implementierung, verwendet ähnliche Primitive wie CML. Es existieren eine sequentielle, parallele und auch eine verteilte Implementierung von Poly/ML. Anders als CML kommt Poly/ML aber nicht ohne Eingriffe in die tieferen Schichten des Laufzeitsystems aus. Die Erzeugung von Threads ist in Poly/ML zudem teurer als in CML, da Poly/ML nicht die CPS Technik verwendet und daher einen Stack für jeden Thread allozieren muß.

FACILE [TLK97] verwendet wie CML und Poly/ML Kanäle als zentrales Kommunikationsmit-

tel. Wie CML so basiert auch FACILE auf den Quellen von SML/NY, nimmt aber anders als CML aus Performanzgründen auch Eingriffe am Laufzeitsystem vor. FACILE fokussiert auf eine verteilte (d.h. Möglichkeit der Kommunikation von Threads, die auf unterschiedlichen Rechnern situiert sind) Implementierung von SML und führt aus diesem Grund Nebenläufigkeit in die Sprache ein.

11.3 WAM

Das erste Sprachdesign von Oz war noch stark von den Ideen des logischen Programmierens beeinflußt. Die erste Implementierung für die logische Programmiersprache Prolog, deren Performanz mit vergleichbaren Sprachen konkurrieren konnte, wurde von Warren vorgestellt [War77, War83]: die Warren Abstract Machine (WAM). Folglich war das erste Design der Maschine für Oz zunächst auch von der WAM geprägt. Auch wenn Oz aus heutiger Sicht nur wenige Gemeinsamkeiten mit Prolog verbinden und Oz primär als funktionale, denn als logische Programmiersprache gesehen werden muß, gehen verschiedene Techniken und Ideen bei der Implementierung noch auf die WAM zurück. Wir sind im Verlauf dieser Arbeit bereits an verschiedenen Stellen auf Gemeinsamkeiten und Unterschiede eingegangen und wollen diese an dieser Stelle noch einmal kurz zusammenfassen:

Rücksetzen Zur Implementierung von Backtracking verwendet die WAM die Technik, daß alle (relevanten) Änderungen ab einem Wahlpunkt gemerkt werden, damit sie gegebenenfalls wieder rückgängig gemacht werden können. Zwar unterstützt Oz kein Backtracking, dennoch konnte diese Technik bei der Implementierung von (tiefen und flachen) Wächtern eingesetzt werden: so werden beim Subsumptionstest eines Wächters Variablenbindungen vorgenommen, die dann später eventuell wieder rückgängig gemacht werden müssen (vgl. Abschnitt 3.9).

Unifikation Die Techniken zur Behandlung von Unifikation, wie beispielsweise Lese-/Schreib-Modus und Argumentzeiger, wurden aus der WAM übernommen (vgl. Abschnitt 4.3).

Datenstrukturen Die Darstellung von Datenstrukturen wie Tupel und die Verwendung markierter Referenzen (vgl. Abschnitt 7.2) wurde aus der WAM übernommen.

Logische Variablen Die Darstellung logischer Variablen unterscheidet sich von der der WAM (vgl. Abschnitt 7.4.7): die WAM verwendet eine Marke REF für logische Variablen. Ungebundene Variablen werden als Selbst-Referenzen dargestellt. In Mozart unterscheiden wir zwischen Referenzen auf Variablen (REF) und der Variablen selbst (VAR). Zudem verwendet Mozart aus Optimierungsgründen unterschiedliche Darstellungen von Variablen (Abschnitt 7.4.6).

Maschinenarchitektur Wie die WAM so verwendet auch Mozart eine registerbasierte Maschinenarchitektur im Gegensatz zu vielen anderen Implementierungen, die auf einer Kellermaschine beruhen (vgl. Abschnitt 5.1).

11.4 AKL

Für die Implementierung lokaler Berechnungsräume, die in dieser Arbeit ausgeklammert wurden und in [MSS95, Sch99] näher beschrieben sind, gingen wichtige Impulse von *AKL* (Andorra Ker-

nel Language) [Jan94, HJP92, JH91, Har90] aus. Als Oz sich noch in der Designphase befand, lag dort bereits eine funktionsfähige prototypische Implementierung vor.

AKL ist zwar auch nebenläufig, unterscheidet sich in diesem Punkt aber radikal von Oz. Neue Threads werden hier sehr feinkörnig und implizit erzeugt, nicht zuletzt weil hier eine parallele Implementierung der Sprache [Mon97] eine wichtige Rolle spielte. Zudem garantiert AKL anders als Oz keine Fairneß durch Preemption.

Die Performanz der Implementierung von AKL konnte nie wirklich überzeugen. So berichtet Johan Montelius [Mon97] von einem Verlust um einem Faktor zwei bis vier gegenüber Sicstus Prolog für die sequentielle Implementierung namens Agents; die parallele Version Penny ist gegenüber Agents noch einmal geringfügig langsamer. Techniken zu einer effizienteren Implementierung von AKL [Bra95] wurden nicht in Agents umgesetzt.

Die Entwicklung von AKL wurde dann zugunsten der gemeinsamen Weiterentwicklung von Oz eingestellt.

11.5 Erlang

Erlang [AVWW96, Arm97] ist eine nebenläufige, funktionale, dynamisch getypte Programmiersprache. Erlang wurde von Ericsson entwickelt und wird unter anderem zur Implementierung zuverlässiger Echtzeit-Systeme (wie zum Beispiel Telefonvermittlungsstellen) eingesetzt [Wik96].

Nebenläufigkeit wird durch explizite Erzeugung von Threads (die in Erlang als Prozesse bezeichnet werden) ermöglicht. Threads können first-class referiert werden; Kommunikation erfolgt durch asynchrones Senden einer Nachricht an einen Thread. Die Einrichtung mehrerer Kommunikationskanäle zwischen Threads ist somit nicht möglich. Das synchrone Empfangen einer Nachricht erfolgt durch eine `case`-artige Anweisung; dabei werden sukzessive alle noch nicht verarbeiteten Nachrichten geprüft, bis die erste gefunden wird, für die ein Wächter erfüllt ist. Wird keine passende Nachricht gefunden, wird der Thread suspendiert.

Zur Implementierung von Erlang wurde das Design zweier sequentieller virtueller Maschinen publiziert: JAM [JLAW92] ist eine emulatorbasierte Maschine, während TEAM/BEAM [Hau94] eine effiziente native Implementierung (durch Übersetzung nach C) der Sprache ist.

Die Sprache selbst und insbesondere auch deren Implementierungen sind von Prolog und damit der WAM beeinflußt: das betrifft zum Beispiel die Darstellung der Datenstrukturen (Tupel, markierte Referenzen) oder die Übersetzung von Funktionen, die über eine klausel-artige Syntax definiert werden.

Im Design beider Maschinen sind für jeden Thread jeweils ein eigener Keller und vor allem auch eine eigene Halde vorgesehen, die aber dennoch initial nur wenig Speicher (wenige hundert Worte) benötigen. Die Existenz separater Halden hat den Nachteil, daß beim Schicken von Nachrichten, diese stets von der Halde eines Threads auf die eines anderen vollständig kopiert werden müssen. Andererseits ermöglicht dies eine leichtere Verteilung von Threads auf unterschiedliche Rechner. Zudem benötigt jede einzelne Speicherbereinigung, die ja nun pro Thread durchgeführt werden kann, weniger Zeit als eine globale Speicherbereinigung, was im Kontext von Echtzeitsystemen von Bedeutung (aber nicht unbedingt ausreichend) ist.

Der Scheduler garantiert Fairneß, allerdings wird das Zeitintervall eines Threads nicht wie in Mozart durch Verwendung eines Signal-Timers realisiert sondern durch einen Zähler.

Die JAM ist eine Stack-Maschine, während TEAM/BEAM durch ein registerbasiertes Design besser auf die Erzeugung von nativem Code für moderne Prozessoren zugeschnitten ist. Hausman gibt in [Hau94] einen Performanzgewinn in der Laufzeit von einfachen Benchmarks von etwa einer Größenordnung im Vergleich zur JAM an. Im Vergleich zu anderen nativen Systemen schneidet TEAM bei einem von Fließkomma-Operationen dominierten Benchmark vergleichsweise schlecht ab: gemäß [HFA+96] fällt Turbo-Erlang etwa im Vergleich zu SML/NJ und ML-Works um einen Faktor 5 zurück.

11.6 Concurrent Constraint Sprachen

Ausgehend von der Relational Language [CG81] erlang die Familie der nebenläufigen logischen (concurrent constraint) Programmiersprachen mit der starken Vermarktung von Concurrent Prolog von Shapiro [Sha83] besonderes Interesse. Die Relational Language ging dann über in PARLOG [Gre87] und später in Strand [FT89]. GHC (guarded horn clauses) [Ued85] versuchte Schwächen von PARLOG und Concurrent Prolog auszumerzen.

Alle diese Sprachen waren mehr oder weniger stark von dem Gedanken einer parallelen Implementierung inspiriert.

Allen Sprachen gemeinsam ist die Idee, eine Prozedur `p` über eine oder mehrere Klauseln der Form

```
p :- g | b.
```

zu definieren. Dabei ist `g` der Wächter und `b` der Rumpf einer Klausel. Bei einem Aufruf von `p` wird geprüft, ob der Wächter einer der Klauseln erfüllt ist; in diesem Fall wird der Aufruf dann zum Rumpf der entsprechenden Klausel reduziert. Die Definition, was unter dem Erfülltsein eines Wächters zu verstehen ist, war von adhoc Konstruktionen (z.B. read-only Variablen in Concurrent Prolog) geprägt, die von dem intendierten operationalen Verhalten und dem implementierungstechnisch Machbaren gesteuert waren.

Erst Maher [Mah87] definierte die Erfülltheit eines Wächters formal über logische Implikation (Entailment).[1] Beeinflußt von Maher und den Arbeiten von Jaffer und Lassez [JL87] führte später Saraswat [Sar89] das cc (concurrent constraint) Gerüst ein, das auf zwei Grundoperationen ask (= Entailment) und tell (Hinzufügen eines Constraints zum Speicher) basiert.

In den Wächtern ließen die Sprachen zunächst beliebige Atome (= Prozeduraufrufe) zu. Man mußte jedoch bald erkennen, daß sich die Sprachen in dieser Allgemeinheit nicht effizient implementieren ließen. Daher schränkte man später die Form der Wächter deutlich ein: es wurden nur noch sogenannte *flache* Wächter zugelassen, die nur bestimmte vordefinierte Prozeduren (wie Typtests, arithmetische Vergleiche oder Gleichungen) enthalten durften. Da die Sprachen Gegensatz zu Oz eine einfache klauselbasierte Syntax verwenden, bedingte dies oft eine umständliche Umformulierung von Programmen, um Prozeduraufrufe aus Wächtern heraus zu falten. Durch die Beschränkung auf flache Wächter konnte man deutliche Performanz-Gewinne erzielen (allerdings blieb beispielsweise Flat Concurrent Prolog immer noch etwa um einen Faktor 5 hinter Prolog zurück [HS89, Deb93]). Im Gegensatz dazu läßt Oz beliebige Ausdrücke in Wächtern von

[1] Genauer: eine Klausel wird dann ausgewählt, wenn ihr Wächter entailed ist, oder wenn alle anderen Klauseln disentailed sind. Maher definierte somit als erster das Andorra-Prinzip [War87], von dem Sprachen wie Andorra-I [CWY91] und AKL ausgingen.

Konditionalen zu. Allerdings kann der Compiler für eine bestimmte Klasse von Konditionalen deutlich effizienteren Code erzeugen; die Verfahren die hierzu eingesetzt werden, wurden in dieser Arbeit bei der Implementierung des Case eingehend vorgestellt. Diese flachen (aus Sicht der Implementierung) Wächter decken in der Praxis nun fast alle Vorkommen von Konditionalen ab. Tiefe Wächter werden in Oz fast ausschließlich für Suche benötigt [Sch99]. Somit erlaubt Mozart die Verwendung von tiefen Wächtern ohne Einschränkung, ohne daß dadurch die Verwendung von flachen Wächtern in der Praxis mit Performanzeinbußen bezahlt werden muß.

Auch die aktuellen Vertreter der cc Sprachen wie KL1 [Chi92] (das aus Flat GHC hervorging) oder Janus [SKL90] machen wie ihre Vorgänger noch Einschränkungen in der Verwendung von Wächtern. Janus beispielsweise schränkt mit der two-occurence-restriction die Art der Verwendung von Variablen deutlich ein; KL1 kann nicht wirklich Entailment eines Wächters entscheiden, weil hier der Versuch der Bindung einer globalen Variable zur Suspension führt. Wie wir in Abschnitt 3.9 ausführlich diskutiert haben, ist es zur Entscheidung von Entailment nötig, während der Ausführung eines Wächters globale Variablen temporär zu binden. Zudem muß die Bindung lokaler Variablen gesondert behandelt werden. Betrachtet man beispielsweise die Definition eines exklusiven Oders in L

```
fun xor(x,y) =
  case [x,y] of
    [true,false] => true
  | [false,true] => true
  | _            => false
```

so kann in L folgender Ausdruck

```
  val x = lvar();
  xor(x,x)
```

direkt zu `false` reduzieren. In Janus ist eine Formulierung von `xor` in der obigen Form verboten, in KL1 suspendiert die Auswertung. Die Performanz von Sprachen wie AKL, die Entailment auch für tiefe Wächter entscheiden konnten, konnte wie in Abschnitt 11.4 erwähnt, nicht überzeugen. Dagegen erreichen die Implementierungen von Janus [GDBD92] (jc) und KL1 [CFS94] (KLIC), die beide nach C übersetzen, respektable Ergebnisse; sie sind etwa doppelt so schnell wie natives Sicstus Prolog, was allerdings auch an Eigenschaften der Sprachen selbst liegt (Janus etwa hat keine logischen Variablen) und der Tatsache, daß die Implementierungen durch einen feinkörnigeren Befehlssatz besser auf die Erzeugung von nativem Code abgestimmt sind.

Kapitel 12

Zusammenfassung und Ausblick

In dieser Arbeit haben wir Design, Implementierung und Evaluierung einer virtuellen Maschine für die Kernsprache von Oz vorgestellt.

Hierzu haben wir die Sprache L eingeführt, die (um dem Leser einen möglichst schnellen und einfachen Einstieg zu erlauben) an SML angelehnt ist: L übernimmt die Syntax und einen Teil der Datenstrukturen aus SML. Die wichtigsten Erweiterungen von L gegenüber SML sind: logische Variablen, Threads und Synchronisation.

Die Semantik von L erklärten wir informell am Modell eines Graphen. Aus dem Graphenmodell entwickelten wir dann das Modell einer virtuellen Maschine für L. Wir gingen schrittweise in unterschiedlichen Abstraktionsebenen vor. Wir beschrieben zunächst ein einfaches Grundmodell, das die wichtigsten Ideen zur Realisierung von Threads, Suspension, logischen Variablen und Case wiedergibt. Darauf aufbauend gingen wir auf wichtige Optimierungen dieses Modells ein und beschrieben Techniken wie zum Beispiel zur Code-Spezialisierung und -Instantiierung, zur Optimierung der Unifikation, zur effizienten Behandlung von Konstruktoren, Case und Funktionen.

In einem weiteren Verfeinerungsschritt gingen wir dann auf wichtige Aspekte der Implementierung ein. Wir beschrieben Techniken zur effizienten Implementierung von Emulatoren, zur Speicherverwaltung, zur Darstellung von Datenstrukturen, zu Threads und nativen Funktionen.

Wir haben Mozart, die Implementierung von Oz, mit den Implementierungen anderer Sprachen verglichen. Unter den Emulatoren nimmt Mozart eine Spitzenstellung ein: es ist kompetitiv mit den schnellsten Emulatoren für statisch getypte Sprachen (wie OCAML und Java). Mozart ist in der Regel deutlich schneller als Sicstus Prolog und Erlang, Sprachen, die von ihrem Charakter her (dynamische Typisierung, Nebenläufigkeit, logische Variablen) eher dem von Oz ähneln als etwa die ML Dialekte. Im Vergleich zu nativen Systemen liegt Mozart zwar bei arithmetischen Benchmarks um bis zu einer Größenordnung zurück; bei den primären Anwendungen von Oz wie symbolischen Berechnungen schmilzt dieser Abstand auf einen Faktor von selten mehr als zwei; in bestimmten Fällen kann Mozart hier sogar schneller sein. Vor allem die Implementierung von Threads in Mozart stellt sich als sehr effizient und leichtgewichtig heraus, während etwa die Java-Implementierungen hier völlig versagen.

Schließlich haben wir anhand von größeren realen Applikation das dynamische Verhalten von typischen Oz-Programmen aus unterschiedlichen Anwendungsbereichen untersucht und Profile zur Struktur des Haldenspeichers, der Verwendung von Funktionen, Unifikation und Threads

erstellt. Diese Zahlen belegen die Effektivität der in dieser Arbeit vorgestellten Techniken in der Praxis, zeigen aber auch auf, daß an bestimmten Stellen noch Bedarf für Verbesserungen besteht.

Ausblick

Die Entwicklung von Oz wird zur Zeit vehementer vorangetrieben denn je. Neben der Gruppe in Saarbrücken arbeiten die Gruppen um Seif Haridi am SICS und Peter Van Roy an der Universität Louvain aktiv an der gemeinsamen Weiterentwicklung der Sprache mit.

Die offizielle Freigabe von Mozart steht kurz bevor, gleichzeitig wird aber schon intensiv an neuen Ideen gearbeitet, die sowohl das Design der Sprache selbst als auch deren Implementierung betreffen. Hier geht es, um nur ein Beispiel zu nennen, um eine Ausrichtung auf SML (durch Ersetzen der gewöhnungsbedürftigen Syntax von Oz) gepaart mit einem Redesign des Objektsystems [SR98].

Wir wollen im folgenden kurz verschiedene Punkte erörtern, die aus Sicht der Implementierung, auf die wir uns ja in dieser Arbeit konzentriert haben, für die Zukunft interessant und erfolgversprechend erscheinen.

Die Arbeiten an der Maschine und dem Laufzeitsystem sind davon geprägt, orthogonale Teile heraus zu faktorisieren, was auch weitgehend gelungen ist. So stellt sich deren Architektur so dar, daß im Kern die eigentliche virtuelle Maschine mit ihrem Instruktionssatz und den zentralen Datenstrukturen steht, während viele Erweiterungen wie verschiedene Constraintsysteme, Verteilung oder Suche als orthogonale Erweiterungen mit vergleichsweise kleinen Schnittstellen daran angekoppelt sind. Dieses Zentrum der Maschine stellt sich nun aber als recht stabil dar, an dem selten Änderungen vorgenommen werden, es sei denn es handelt sich um Vereinfachungen und Bereinigungen, die durch Änderungen an der Peripherie bedingt sind. Es wird aber nur sehr selten eine Änderung am Instruktionssatz oder der Darstellung zentraler Datenstrukturen notwendig. Das zeigt, daß das Design des Herzens der Maschine universell genug ist, den meisten Bedürfnissen Rechnung zu tragen.

Während somit aus heutiger Sicht der Kern der Maschine weitgehend stabil ist, erscheint aus Implementierungssicht eine Erweiterung des Compilers vielversprechend. Momentan wird die Basissprache von Oz gut auf den Instruktionssatz der Maschine abgebildet. Das betrifft einen recht hohen Anteil der in der Praxis verwandten Konstrukte. Allerdings kann es gerade dem naiven Benutzer immer wieder passieren, daß er Konstrukte verwendet, die suboptimal behandelt werden. Das liegt an der Komplexität der Sprache Oz selbst, die sehr viele eingebaute Datenstrukturen zur Verfügung stellt und durch eine reiche Syntax viele Varianten der Formulierung zuläßt. Fast immer ist hier aber ein Umformulieren möglich, und in der Regel ist dies auch mit wenigen Handgriffen zu erledigen, was wieder darauf schließen läßt, daß die Maschine selbst flexibel und ausreichend leistungsfähig ist. Vieles davon sollte aber eigentlich der Compiler dem Benutzer abnehmen. So scheint die Integration fortgeschrittener Techniken (wie z.B. Inlining, Fold/Unfold-Transformationen, Herausfaktorisieren von Konstanten aus Funktionsrümpfen, insbesondere Elimination von geschachtelten Funktionsdefinitionen, Code-Instantiierung, abstrakte Interpretation im allgemeinen) in den Compiler ein vielversprechendes Betätigungsfeld, das deutliche Gewinne erwarten läßt. Die Anwendung dieser Techniken ist für andere Sprachen zum Teil gut verstanden; hier bleibt zu klären inwieweit sie sich auch auf Oz übertragen lassen und umgekehrt welche neuen Techniken sich nur im Zusammenhang mit Oz anwenden lassen. Hier muß auch nicht notwendig eine feste Integration in den Compiler erfolgen; ähnlich wie Mixtus

für Sicstus Prolog [Sah91] könnte in einer ersten Näherung ein externes Werkzeug eine Transformation auf Quellsprachenebene durchführen.

Interessant ist sicher auch eine Exploration der Möglichkeiten nativer Codeerzeugung, da der Instruktionssatz der Maschine schon über längere Zeit hinweg stabil ist. Hier scheint eine Übersetzung nach C, die auch viele andere Systeme verwenden, für ein erstes Ausloten der Möglichkeiten sinnvoll. Wie unsere Benchmarks ergeben haben, wirkt sich die Tatsache, daß Oz nebenläufig ist, logische Variablen verwendet und dynamisch getypt ist, nicht signifikant auf die Ausführungsgeschwindigkeit aus, wenn man sie mit Emulatoren für Sprachen vergleicht, die diese performanzhemmenden Spracheigenschaften nicht haben. Will man ähnlich gute Ergebnisse bei der nativen Codeerzeugung für Oz erzielen, so muß man dann aber vor allem an der Entwicklung geeigneter Compilationstechniken arbeiten. Insbesondere wird dies dann sicher auch zu einer Verfeinerung des Designs des Instruktionssatzes oder auch der Architektur der Maschine führen.

Ein wesentlicher Fortschritt von Mozart gegenüber DFKI Oz besteht in der Möglichkeit der verteilten Programmierung: Mozart bietet die Illusion eines gemeinsamen Speichers, so daß Oz Prozesse, die auf unterschiedlichen Rechnern laufen, transparent auf gemeinsame Datenstrukturen zugreifen können. Das beinhaltet nicht nur den Austausch von Zahlen, Listen oder Funktionen. Auch zustandsbehaftete Datenstrukturen wie Zellen, Objekte und logische Variablen können transparent benutzt werden: mobile Objekte wandern so beispielsweise im Netz und kommen automatisch immer zu der Lokation, auf der sie gerade benötigt werden; bei stationären Objekten dagegen werden die Nachrichten migriert. Die Protokolle, die hierzu entwickelt wurden, garantieren, daß das Verhalten eines verteilten Systems dem des zentralisierten entspricht.

Die erste Release von Mozart steht zwar kurz bevor, aber es besteht noch ein breites Betätigungsfeld für die Implementierung. So existieren bisher nur kleinere Programme, um die Möglichkeiten verteilter Programmierung in Oz auszuloten. Unklar ist daher, welche Anforderungen größere realistischere verteilte Anwendungen stellen, zum Beispiel ob die in der Implementierung verwandten Protokolle und Algorithmen einen hinreichend schnellen Datenaustausch erlauben. Weiter können momentan zwar ein Großteil, aber noch nicht alle Werte (z.B. Threads, Berechnungsräume, Constraint-Variablen) verteilt werden. Auch wird noch diskutiert, ob eine Verteilung aller Datenstrukturen überhaupt sinnvoll ist oder ob ein minimalistischeres Modell, bei dem man Aufwand und Komplexität spart, nicht sinnvoller ist. Weitere offenen Fragen drehen sich um die Behandlung von Failure im Netzwerk oder die Perfektionierung der verteilten Speicherbereinigung.

Man sieht, für die Weiterentwicklung von Oz besteht sowohl seitens des Sprachdesigns als auch seitens der Implementierung auch in Zukunft ein breites Betätigungsfeld.

Anhang

Anhang A

Die virtuelle Maschine für L

Wir geben im folgenden eine kurze Zusammenfassung über die virtuelle Maschine für L, indem wir zunächst die verschiedenen Adressierungsarten zusammenfassen, danach die virtuellen Maschinenregister und schließlich die Instruktionen der Maschine zusammenstellen.

A.1 Adressierungsarten

Tabelle A.1 faßt die Adressierungsarten der Maschine zusammen.

Die *X-Register* werden pro Thread alloziert. Sie werden zum Zwischenspeichern temporärer Werte oder für kurzlebige Variablen verwandt. Zudem werden sie für die Parameterübergabe bei der Funktionsapplikation und die Rückgabe von Werten bei Funktionsaufrufen verwandt.

Die *L-Register* (L = lokal) werden pro Funktionsinkarnation angelegt und nehmen die lokalen Variablen einer Funktion auf. Der Compiler sorgt dafür, daß der Code, der für den Rumpf einer Funktion f erzeugt wird, bei Bedarf bei jedem Aufruf von f einen neuen Satz L-Register hinreichender Größe alloziert und diese entsprechend füllt.

Die *G-Register* (G = global) dienen zur Adressierung der freien Variablen einer Funktion und werden bei der Erzeugung einer Funktion angelegt.

Die *V-Register* (V = virtuell) dienen zur Adressierung von konstanten Werten, die sich über die gesamte Lebensdauer eines Programmes hinweg nicht verändern werden. Referenzen auf V-Register können bereits vom Lader aufgelöst werden.

Tabelle A.1 Registerarten der Maschine

Name	Lebensdauer	Schreibweise	Typische Verwendung
X	Thread	$\mathtt{X}_i$	temporäre Werte, Parameterübergabe, etc.
lokal	Funktionsinkarnation	$\mathtt{L}_i$	lokale Variablen einer Funktion
global	Funktion	$\mathtt{G}_i$	globale (= freie) Variablen einer Funktion
virtuell	Programm	$\mathtt{V}_i$	konstante Argumente

A.2 Register

Die Maschine verwendet die folgenden Register:

`X`
: Das Register `X` ist ein Zeiger auf ein Feld der Länge n zur Darstellung der n festen X-Register $\mathtt{X}_0, \ldots, \mathtt{X}_{n-1}$ eines Threads.

`PC`
: Das Register `PC` repräsentiert den *Programmzähler*, der auf die Instruktion verweist, die die Maschine gerade ausführt.

`L`
: Verweis auf die lokalen Variablen der gerade in Ausführung befindlichen Funktion.

`G`
: Verweis auf die globalen Variablen der gerade in Ausführung befindlichen Funktion.

`runqueue`
: Das Register verweist auf die Schlange, die alle ausführbaren Threads (ausschließlich des gerade laufenden Threads) enthält.

`running`
: Das Register verweist auf den gerade in Ausführung befindlichen Thread.

`SP`
: Kellerzeiger, der auf den obersten Eintrag von `running` verweist.

`mode`
: Das Modusregister dient zur optimierten Behandlung von Tupeln im Wächter eines Case. Es kann die Werte `read` oder `write` annehmen, je nachdem ob die Maschine mit dem Matching eines Tupels oder dem Aufbau eines neuen Tupels auf der Halde beschäftigt ist.

`ap`
: Der Argumentzeiger dient zum Aufbau und Matching von Tupeln. Er verweist auf das nächste zu behandelnde Argument eines Tupels.

`caseStart`
: Das Register wird zur Implementierung von Case benötigt. Es verweist auf den Anfang der Instruktionen für den Guard eines Case.

`heapSave`
: Das Register dient zum effizienten Test auf Lokalität einer Variablen bei der Ausführung eines Wächters eines Case.

`trail`
: Das Register verweist auf die Spur, einen Stapel, der temporäre Variablen-Bindungen aufnimmt,

`suspVars`
: Das Register dient zur Behandlung der Suspension von nativen Funktionen. Es verweist auf die Liste aller Variablen, auf die eine native Funktion suspendiert.

`heapTop`, `heapEnd`, `freelist`
: Register zur Verwaltung der Halde und der Freispeicherliste.

A.3 Instruktionen

Wir fassen im folgenden die Instruktionen der Maschine zusammen, indem wir sie nach Art ihrer Verwendung gruppieren.

A.3.1 Prelude

Die Prelude-Instruktionen werden bereits vom Lader ausgeführt und dienen zum Aufbau von Werten auf der Halde auf die die Instruktionen des zu ladenden Programmes dann direkt verweisen.

`vnewInt` $\mathtt{V}_i$ n
: Die Instruktion erzeugt einen neuen Zahl-Knoten mit Wert n im Speicher und legt eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$ ab.

`vnewVar` $\mathtt{V}_i$
: Erzeugt einen neuen Variable-Knoten im Speicher und speichert eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$.

`vnewCon` $\mathtt{V}_i$
: Erzeugt einen neuen Konstruktor-Knoten im Speicher und speichert eine Referenz auf diesen Knoten im Register $\mathtt{V}_i$.

`export` $\mathtt{V}_i$ c
: Trägt in die Exporttabelle unter der Zeichenkette c den Inhalt von $\mathtt{V}_i$ ein. Wenn bereits ein Eintrag für c existiert, wird dieser überschrieben.

`import` $\mathtt{V}_i$ c
: Lädt den Knoten, der unter der Zeichenkette c in der Exporttabelle gespeichert ist, in V-Register $\mathtt{V}_i$. Wenn kein Eintrag für c gefunden wird, erfolgt eine Fehlermeldung.

`vmakeHT` V n
: Erzeugt eine neue Hashtabelle zur indizierten Übersetzung eines Case der Größe n und legt eine Referenz darauf im V-Register V ab.

`vaddHTCon` V V' L
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle und speichert in dieser unter dem Konstruktor aus V' die Marke L.

`vaddHTInt` V `n` L
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle und speichert in dieser unter der Zahl n die Marke L.

`vaddHTTuple` V V' n L
: Die Instruktion speichert in der Hashtabelle aus V, daß für ein Tupel mit Marke V' und Arität n zur Marke L zu springen ist.

A.3.2 Zellen, Konstruktoren, logische Variablen

`newRef` R R'
: Erzeugt eine neue Zelle im Speicher und legt eine Referenz darauf in Register R ab. Inhalt der Zelle ist der Knoten, auf den R' verweist.

`exchange` R R' R''
: Die Instruktion wartet (das heißt der ausführende Thread wird solange suspendiert), bis R eine Zelle referiert. Danach wird in R' der aktuelle Inhalt der Zelle abgelegt und der neue Inhalt der Zelle zu R'' gesetzt.

`newCon` R
: Erzeugt einen neuen Konstruktor-Knoten im Speicher und legt eine Referenz darauf in Register R ab.

`newVar` R
: Erzeugt einen neuen Variable-Knoten im Speicher und legt eine Referenz darauf in Register R ab.

A.3.3 Tupel

`newTuple` R R' n
: Die Instruktion erwartet in R' einen Konstruktor-Knoten und erzeugt ein neues Tupel mit Marke R' und n uninitialisierten Argumenten und legt einen Verweis darauf in R ab.

`newList` R
: Erzeugt einen neuen `LIST`-Knoten und legt eine Referenz darauf nach R. Zudem wird `ap` auf das erste Argument der Liste gesetzt.

`setArg` R
: Schreibt den Inhalt von R in das Argument eines Tupels, auf den der Argumentzeiger `ap` verweist und inkrementiert `ap`, so daß `ap` auf das folgende Argument verweist.

`setVarArg` R
: Erzeugt in dem Argument eines Tupels, auf den der Argumentzeiger `ap` verweist, eine neue Variable-Zelle. Im Register R wird eine REF-Zelle abgelegt, die auf dieses Argument verweist. Anschließend wird `ap` inkrementiert.

`setVoid` n
: Die Instruktion schreibt neue Variablen in die nächsten n Argumentpositionen, auf die `ap` verweist. Anschließend wird `ap` um n Argumentpositionen erhöht.

A.3.4 Unifikation

`unify` R R'
: Die Instruktion unifiziert die Inhalte die Register R und R'.

`unifyConst` R V
: Die Instruktion dereferenziert R. Falls R eine Variable enthält, wird diese an den Inhalt des V-Registers V gebunden. Andernfalls wird getestet, ob R und V auf den selben Knoten im Speicher verweisen.

`unifyInteger` R V
: Die Instruktion erwartet im V-Register V eine Referenz auf einen Zahlknoten. Falls R nach Dereferenzierung eine Variable enthält, wird diese an den Inhalt von V gebunden. Andernfalls wird getestet, ob R auf einen Zahlknoten verweist, der den gleichen Wert wie V hat.

A.3.5 Arithmetik

`plus` R R' R'', `minus` R R' R'', `times` R R' R'', `div` R R' R'', `less` R R' R''
: Die Instruktion wartet, bis R und R' auf eine Zahl verweisen, und legt dann das Ergebnis der arithmetischen Operation in R'' ab.

A.3.6 Threads

`spawn` R
: Die Instruktion wartet, bis das Register R eine nullstellige Funktion f enthält. Sie erzeugt dann einen neuen Thread T, der die Auswertung der Applikation von f durchführt.

A.3.7 Funktionen

`fun` R n k S
: Erzeugt einen neuen Funktions-Knoten mit Arität n und k freien Variablen und legt eine Referenz darauf in R ab. Die Instruktion erwartet die Werte der globalen Variablen in den Registern $\texttt{X}_0$ bis $\texttt{X}_{k-1}$. S ist eine Referenz auf das Segment, das den Code für den Rumpf der Funktion enthält.

`funCopy` R n k S
: Erzeugt analog der Instruktion `fun` einen neuen Funktions-Knoten mit Arität n und k freien Variablen und legt eine Referenz darauf in R. Anders als `fun` wird aber auch zusätzlich eine *Kopie* des Codesegmentes S angefertigt und im neu erzeugten Funktions-Knoten gespeichert.

`apply` R n
: Die Instruktion prüft zunächst, ob eine Signalbehandlung notwendig ist. Wenn ja wird der aktuelle Thread gestoppt und die Signalbehandlung durchgeführt. Andernfalls wird gewartet, bis R einen Konstruktor- oder Funktions-Knoten enthält.

  Enthält R einen Konstruktor-Knoten c, dann wird ein neues n-stelliges Tupel mit Marke c erzeugt, dessen i-tes Argument gerade der Inhalt von $\texttt{X}_i$ ist. Das neu erzeugte Tupel wird in $\texttt{X}_0$ abgelegt.

  Enthält R dagegen eine Funktion f mit Arität n, so wird ein neuer *zusätzlicher* Auftrag zur Ausführung des Rumpfes von f auf dem aktuellen Thread erzeugt.

`tailApply` R n
: Endrekursive Version von `apply`.

`fastApply` V
: Die Instruktion erwartet im V-Register V eine Referenz auf einen Funktion-Knoten f.

Sie arbeitet analog zu `apply` ohne allerdings Dereferenzierung, Typtests und Aritätstest durchzuführen.

`fastTailApply` V
Endrekursive Version von `fastApply`.

`applyBuiltin` V
Die Instruktion erwartet im V-Register V eine Referenz auf eine native Funktion f und appliziert diese ohne weitere Dereferenzierungen, Typtests und Aritätstest durchzuführen.

`specApply` V n
Die Instruktion suspendiert auf Determiniertheit von V. Verweist V auf einen n-stelligen Funktion-Knoten, so ersetzt sich selbst dann durch `fastApply` V. Verweist V auf einen Konstruktor, dann ersetzt sie sich durch `newTuple` V n.

`specTailApply` V n
Endrekursive Version von `specApply`.

`allocate` n
Alloziert einen neuen (uninitialisierten) L-Registersatz der Größe n. Der L-Zeiger des aktuellen Auftrags zeigt nach Ausführung der Instruktion auf diesen Registersatz, der alte Wert des L-Zeigers geht verloren.

`deallocate`
Gibt den Registersatz frei, auf den der L-Zeiger des aktuellen Auftrags verweist und setzt den L-Zeiger auf **`nil`**.

`return` R
Kopiert den Inhalt des Registers R ins Register $\mathtt{X}_0$. Danach wird der oberste Auftrag des aktuellen Threads T gelöscht und der darunterliegende Auftrag zur Ausführung gebracht. Wenn kein weiterer Auftrag existiert, wird T terminiert und der nächste ausführbare Thread zur Ausführung gebracht.

A.3.8 Case

`guardStart` L
Die Instruktion markiert den Beginn eines Wächters. Die Marke L verweist auf den Beginn des Codes des nächsten Wächters. Die Instruktion setzt das Register `caseStart` auf die aktuelle Instruktion.

`guardEnd`
Die Instruktion markiert das Ende eines Wächters: es wird geprüft, ob die Spur `trail` leer ist. Wenn ja, wird die nächste Instruktion ausgeführt. Wenn die Spur nicht leer ist, wird eine Suspension erzeugt und in die Suspensionsliste aller Variablen x, die die Spur enthält, eingetragen. Der aktuelle Auftrag wird so modifiziert, daß er auf `guardStart` verweist, so daß der suspendierte Wächter nach dem Wecken von neuem ausgeführt wird.

`getInt` R V
Die Instruktion erwartet in V eine Zahl n. Enthält R einen Variable-Knoten v, dann wird v an n gebunden und gegebenenfalls (falls v nicht lokal) auf der Spur gemerkt. Enthält R die Zahl n, wird die nächste Instruktion ausgeführt. Enthält R einen Knoten, der keine

Variable ist und zudem verschieden von n ist, dann wird failure ausgelös, das heißt die Spur wird geleert und zum Code des nächsten Wächters gesprungen.

`getCon` R V
: Die Instruktion arbeitet analog zu `getInt`, und erwartet im Unterschied dazu in V einen Konstruktor.

`getVarArg` R
: Wenn `mode` den Wert `write` hat, arbeitet die Instruktion genau wie `setVarArg`. Andernfalls wird das Argument, auf das `ap` verweist, in R geladen: handelt es sich dabei um eine Variable-Zelle, wird zuerst in R eine REF-Zelle darauf erzeugt.

`moveArg` R
: Wenn `mode` den Wert `write` hat, erzeugt die Instruktion einen neuen Variableknoten und legt eine Referenz darauf sowohl in R ab als auch in dem Argument, auf das `ap` zeigt.
Wenn `mode` den Wert `read` hat, wird das Argument, auf das `ap` zeigt, nach R geladen. Anschließend wird `ap` inkrementiert.

`moveVoid` n
: Wenn `mode` den Wert `write` hat, arbeitet die Instruktion genau wie `setVoid`. Andernfalls wird `ap` lediglich um n Argumentpositionen erhöht.

`getTuple` R R' n
: Die Instruktion erwartet in R' einen Konstruktor-Knoten. Zunächst wird der Inhalt von R geprüft: enthält R eine Variable, dann wird ein neues Tupel mit n uninitialisierten Argumenten und mit Marke R' erzeugt und an R gebunden. In R wird eine Referenz auf dieses neu erzeugte Tupel abgelegt. `mode` wird auf `write` gesetzt.
Enthält R keine Variable, wird geprüft, ob R ein Tupel t mit Marke R' und Breite n enthält, wenn nicht, wird failure ausgelöst; wenn ja, wird in `mode` wird auf `read` gesetzt.
In beiden Fällen wird `ap` auf das erste Argument des Tupels gesetzt.

`getList` R
: Arbeitet analog zu `getTuple` für Listen.

`testConst` R V L
: Die Instruktion erwartet im V-Register V einen (dereferenzierten) Verweis auf einen Konstruktor, eine Zelle oder eine Funktion. R wird dereferenziert. Falls R eine logische Variable enthält, wird auf dieser Instruktion suspendiert. Andernfalls wird geprüft, ob R auf den selben Knoten im Speicher wie V verweist. Falls ja, wird die nächste Instruktion ausgeführt, falls nein wird zur Marke L gesprungen.

`testInteger` R V L
: Diese Instruktion arbeitet analog zu `testConst`. Sie erwartet in V allerdings einen Zahl-Knoten, dessen Inhalt mit R verglichen wird.

`testTuple` R f n L
: Falls R nach Dereferenzierung eine Variable enthält, wird suspendiert. Andernfalls wird geprüft, ob R ein Tupel mit Marke f und n Argumenten enthält. Falls nein wird zur Marke L gesprungen. Falls ja, werden `mode` auf `read` und `ap` auf das erste Argument gesetzt, danach wird die nächste Instruktion ausgeführt.

`testList` R L
: Analog zu `testTuple` für Listen.

`loadArg` R
: Das Argument, auf das `ap` zeigt, wird nach R geladen.

`testBool` R L
: Falls R nach Dereferenzierung eine Variable enthält, wird suspendiert. Andernfalls wird geprüft, ob R den Wert **`true`** hat. Falls ja, wird die nächste Instruktion ausgeführt, falls R den Wert **`false`** enthält, wird zur Marke L gesprungen. Andernfalls wird eine Fehlermeldung ausgegeben.

`switch` R V
: Die Instruktion erwartet im V-Register V eine Referenz auf eine Hashtabelle H. Falls R eine Variable enthält wird suspendiert. Andernfalls liefert H abhängig vom Inhalt von R eine Codeadresse, zu der dann gesprungen wird.

A.3.9 Sonstige Instruktionen

`move` R R'
: Die Instruktion kopiert den Inhalt von R in das Register R'.

`branch` L
: Unbedingter Sprungbefehl, der die Instruktionen, die an der Marke L beginnen, zur Ausführung bringt.

Anhang B

Applikationen für Testläufe

Zur Analyse sowohl der statischen Struktur als auch des dynamischen Verhaltens von Oz Programmen wurde eine Reihe von Anwendungen herangezogen. Dabei handelt es sich nicht um die immer wieder gleichen kleinen Benchmarkprogramme, die in der Literatur vielfach zum Performanzvergleich verschiedener Systeme herangezogen werden. Diese testen in der Regel nur ganz spezielle Aspekte eines Programmiersystems und geben keinen Aufschluß über das typische Aussehen und Verhalten von „richtigen“ Applikationen.

Daher wurde eine Reihe von Applikationen ausgewählt, die für reale Anwendungen entwickelt wurden und sich zum großen Teil täglich im Gebrauch befinden. Der Quellcode jeder Applikation umfaßt mehrere tausend Zeilen Oz Quellcode; jeder Testlauf beanspruchte mindestens mehrere CPU Sekunden.

Für alle Messungen wurde eine Beta-Version von Mozart 1.0 verwandt.

Bei den verschiedenen Anwendungen handelt es sich im einzelnen um:

Prelude Die Oz Prelude enthält eine ganze Reihe von Modulen und Werkzeugen, die bei Programmentwicklung unter der Oz Programmierschnittstelle OPI zur Verfügung stehen. Das sind zum einen die Standardmodule [HMSW97], die verschiedene Bibliotheksfunktionen, wie beispielsweise die gängigen Listenoperationen, zur Verfügung stellt. Darüberhinaus gehören noch die Schnittstelle zum Grafiksystem und verschiedene Werkzeuge, wie der Browser, Explorer, Panel, Profiler und Ozcar (siehe unten) zur Prelude.

Bei den Testläufen wurde das reine Laden der Prelude für Messungen herangezogen; da eine Komponente erste dann geladen wird, wenn sie zum ersten Mal verwandt wird (*lazy loading*), wurde für die Testläufe das Laden explizit angestoßen. Diese Werte liefern insofern bereits interessante Ergebnisse, da beim Laden nicht nur Prozeduren definiert werden, sondern auch schon erhebliche Rechenarbeit geleistet wird: so werden beim Laden der Prelude circa 450 Klassen definiert, wobei die Routinen zum dynamischen Aufbau der Klassenhierarchie in Oz selbst implementiert wurden [Hen97].

Browser Der Browser [Pop95] ist ein Werkzeug zur grafischen Darstellung des Speichers, der es erlaubt, über eine Reihe von Optionen die Art der Darstellung zu beeinflussen. Für die Testläufe wurde vom Browser ein vollständiger binärer Baum der Tiefe 8 angezeigt, wobei die inneren Knoten aus Tupeln, die Blätter zu gleichen Teilen aus Zahlen, Namen, Atomen, Funktionen und Variablen bestanden. Für die Art der Darstellung wurden die Default-Einstellungen verwandt.

Der Quellcode des Browsers besteht aus 15000 Zeilen Oz Code.

Explorer Beim Explorer handelt es sich um ein interaktives, graphisches Werkzeug, das zur Visualisierung und zum Analysieren von Suchbäumen dient [Sch97]. Der Quellcode umfaßt etwas über 5500 Zeilen Oz Code.

Als Eingabe diente ein einfaches Programm, das zu zwei Finite Domain Variablen `X` und `Y` mit einem Wertebereich von jeweils 1 bis 70 durch einfaches Aufzählen des kompletten Suchbaums alle Lösungen bestimmt, für die $\texttt{X} \in \{1, 30, 40\} \wedge \texttt{Y} \in \{1, 30, 40\}$ gilt.

Der so erzeugte Suchbaum hat eine Tiefe von $2 \times 70 = 140$ und insgesamt 25120 Knoten (wovon $3 \times 3 = 9$ Knoten eine Lösung enthalten).

Gump Bei *Gump* [Kor96, Kor97] handelt es sich analog zu *Lex* und *Yacc* [LMB92] um ein Werkzeug, das ausgehend von einer abstrakten Beschreibungsprache in der Lage ist, den Oz Quellcode für einen Scanner oder Parser zu erzeugen. Der Quellcode von Gump umfaßt ca. 3500 Zeilen.

Für die Testläufe wurde sowohl die Beschreibung des Scanners (in den Tabellen mit *Gump Scanner* bezeichnet) als auch die des Parsers (*Gump Parser*) für SML 97 verwandt.

Compiler Die Produktionsversion des in Oz geschriebenen Compilers von Mozart. Für die Testläufe wurde (wenn nicht anders angegeben) der Browser (siehe oben) übersetzt. Der Quellcode umfaßt ca. 19000 Zeilen.

Ozcar Ozcar ist ein grafischer Debugger für Mozart [Lor98]. Der Quellcode umfaßt 4000 Zeilen.

Spedition Hier wurde die Simulation eines Speditionsszenarios in Oz implementiert. Autonome Agenten verhandeln untereinander dynamisch über die Vergabe und Durchführung von Frachtaufträgen, die über Deutschland verteilt sind.

Der Quellcode umfaßt 3900 Zeilen.

Scheduler Hierbei handelt es sich um eine typische Anwendung aus dem Bereich des Constraint Programmierens. Der Oz Scheduler [Wür96] ist ein Werkzeug zur Lösung von Scheduling Problemen. Für die Testläufe wurde `ABZ6` ein 10×10 Job-Shop Problem aus [AC91] verwandt. Es wurden jeweils zwei verschiedene Testläufe mit diesem Problem durchgeführt: zunächst wurde eine obere Schranke für die optimale Lösung berechnet (in den Tabellen mit *Scheduler (upper)* bezeichnet); danach wurde branch-and-bound verwandt, um eine bessere Lösung zu berechnen, als die, die zuvor gefunden wurde (*Scheduler (prove)*). Während bei letzterem dem reinen Constraintlösen der größte Anteil an der Berechnung zukommt, kommen beim ersten auch andere Aspekte wie zum Beispiel objektorientierte Programmierung zum Tragen.

Der Quellcode umfaßt 6200 Zeilen.

Anhang C

Benchmark-Programme

Wir haben hier alle Programme in der Kodierung für SML zusammengetragen, die in Kapitel 10 für Performanzvergleiche herangezogen wurden. Der Programmcode für die anderen Sprachen kann über

`http://www.ps.uni-sb.de/~scheidhr/thesis/benchmarks`

erreicht werden.

C.1 Takeushi

C.1.1 Takeushi mit ganzen Zahlen

```
fun tak(x,y,z) =
   if y<x then tak(tak(x-1,y,z),tak(y-1,z,x),tak(z-1,x,y))
   else z;
```

C.1.2 Takeushi unter Verwendung von CPS

```
fun cpstakaux(x,y,z,k) =
   if y<x then
      cpstakaux(x-1,y,z,
                fn(v1)=>
                cpstakaux(y-1,z,x,
                          fn(v2)=>
                          cpstakaux(z-1,x,y,
                                    fn(v3)=>cpstakaux(v1,v2,v3,k))))
   else
      k(z);
```

```
fun cpstak(x,y,z) = cpstakaux(x,y,z,fn(a)=>a);
```

C.2 Fibonacci

C.2.1 Fibonacci mit ganzen Zahlen

```
fun fib(n) =
    if (2<n) then fib(n-2)+fib(n-1)
    else 1;
```

C.2.2 Fibonacci mit Fließkommazahlen

```
fun fibf(n:real) =
    if (2.0<n) then fibf(n-2.0)+fibf(n-1.0)
    else 1.0;
```

C.3 Naiv Reverse

```
fun app(nil,ys)   = ys
  | app(x::xr,ys) = x::app(xr,ys);

fun nrev(nil)   = nil
  | nrev(a::bs) = app(nrev(bs),[a]);
```

C.4 Quicksort

C.4.1 Quicksort mit Listen

```
fun quickaux([],cont) = cont
  | quickaux(a::bs,cont) = partition(bs,a,[],[],cont)

and partition ([],a,left,right,cont): int list =
    quickaux(left,a::quickaux(right,cont))
  | partition(x::xs,a,left,right,cont) =
    if x < a then partition(xs,a,x::left,right,cont)
    else partition(xs,a,left,x::right,cont);

fun quick l = quickaux(l,[]);
```

C.4.2 Quicksort mit Listen und higher-order Tests

```
fun quickaux([],cont,cmp) = cont
  | quickaux(a::bs,cont,cmp) = partition(bs,a,[],[],cont,cmp)

and partition ([],a,left,right,cont,cmp): int list =
    quickaux(left,a::quickaux(right,cont,cmp),cmp)
  | partition(x::xs,a,left,right,cont,cmp) =
    if cmp(x,a) then partition(xs,a,x::left,right,cont,cmp)
    else partition(xs,a,left,x::right,cont,cmp);

fun less(x,y) = x<y;

fun quickho(l,cmp) = quickaux(l,[],cmp);
```

C.4.3 Quicksort mit Feldern

```
fun partitionarray1(ar,pivot,pindex,from,to) =
    if from <= to then
        let val old = Array.sub(ar,from)
        in if pivot > old then
            (Array.update(ar,from,Array.sub(ar,pindex));
             Array.update(ar,pindex,old);
             partitionarray1(ar,pivot,pindex+1,from+1,to))
           else
               partitionarray1(ar,pivot,pindex,from+1,to)
        end
    else pindex-1;


fun partitionarray(ar,from,to) =
    let val pivot = Array.sub(ar,from)
        val mid = partitionarray1(ar,pivot,from+1,from+1,to)
    in  (Array.update(ar,from,Array.sub(ar,mid));
         Array.update(ar,mid,pivot);
         mid)
    end;

fun quickarray1(ar,from:int,to:int) =
   if from < to then
      let val mid = partitionarray(ar,from,to)
      in (quickarray1(ar,from,mid-1);
          quickarray1(ar,mid+1,to))
      end
   else ();
```

```
fun quickarray(ar) = quickarray1(ar,0, Array.length(ar)-1);
```

C.5 N-Damen

```
fun no_attackaux(nil,c,y,i) = true
  | no_attackaux(x::xr,c,y,i) =
    (x<>y) andalso
    ((abs (x-y))<>(c-i)) andalso
    no_attackaux(xr,c,y,i+1);

fun no_attack(xs,c,y) = no_attackaux(xs,c,y,1);

fun queensLoop2(ss,c,xs,y,n) =
   if y>n then ss
   else if no_attack(xs,c,y)
        then queensLoop2(app(xs,[y])::ss,c,xs,y+1,n)
        else queensLoop2(ss,c,xs,y+1,n);

fun doFoldL(nil,z,t,c) = z
  | doFoldL(x::xr,z,t,c) = doFoldL(xr,queensLoop2(z,c,x,1,t),t,c);

fun queensLoop1(c,t,i) =
   if c<=t then
         queensLoop1(c+1,t,doFoldL(i,nil,t,c))
   else i;

fun queens(n) = queensLoop1(1,n,[nil]);
```

C.6 Mandelbrot

```
val x_base = ~2.0
val y_base = 1.25
val side = 2.5

val sz = 800
val maxCount = 1024

val delta = side / (real sz)
```

```
fun mandelloop3 (count, z_re:real, z_im:real, c_re:real, c_im:real) =
    if (count < maxCount)
        then
            let
                val z_re_sq = z_re * z_re
                val z_im_sq = z_im * z_im
            in
                if ((z_re_sq + z_im_sq) > 4.0)
                    then count
                else
                    let
                        val z_re_im = (z_re * z_im)
                    in
                        mandelloop3 (count+1,
                                     (z_re_sq - z_im_sq) + c_re,
                                     z_re_im + z_re_im + c_im,
                                     c_re, c_im)
                    end
                end
        else count;

fun mandelloop2(j,c_im,iter) =
    if (j >= sz) then iter
    else
        let val c_re = x_base * (delta + real j)
            val count = mandelloop3(0,c_re,c_im,c_re,c_im)
        in
            mandelloop2(j+1,c_im,iter+count)
        end;

fun mandelloop(i,iter) =
    if (i >= sz) then iter
    else
        let val c_im : real = y_base - (delta * real i)
        in mandelloop(i+1,mandelloop2(0,c_im,iter))
        end;
```

C.7 Symbolische Ableitung

```
datatype expr = Plus  of expr * expr
              | Minus of expr * expr
              | Var of int
              | Const of int
              | Times of expr * expr
              | Div of expr * expr
              | Exp of expr * int
              | Uminus of expr
              | Log of expr;

fun deriv(Var(u),x)     = if u=x then Const(1) else Const(0)
  | deriv(Const(u),x)   = Const(0)
  | deriv(Plus(u,v),x)  = Plus(deriv(u,x),deriv(v,x))
  | deriv(Minus(u,v),x) = Minus(deriv(u,x),deriv(v,x))
  | deriv(Times(u,v),x) = Plus(Times(deriv(u,x),v),Times(u,deriv(v,x)))
  | deriv(Div(u,v),x)   = Div(Minus(Times(deriv(u,x),v),
                                    Times(u,deriv(v,x))),
                              Exp(v,2))
  | deriv(Exp(u,n),x)   = Times(Times(deriv(u,x),Const(n)),Exp(u,n-1))
  | deriv(Uminus(u),x)  = Uminus(deriv(u,x))
  | deriv(Log(u),x)     = Div(deriv(u,x),u);

fun nthderiv(0,exp,x) = exp
  | nthderiv(n,exp,x) =
    nthderiv(n-1,deriv(exp,x),x);
```

C.8 Threads

C.8.1 Erzeugung

```
fun noop (n:int) = ();

fun threadcreate(0) = ()
  | threadcreate(n) =
    let val id = CML.spawn(fn()=>noop(n))
    in  CML.sync(CML.joinEvt(id));
        threadcreate(n-1)
    end;
```

C.8.2 Kommunikation

```
fun reader(0,chan) = CML.send(chan,0)
  | reader(n,chan) =
    let val aux = CML.send(chan,1)
    in  CML.recv(chan);
        reader(n-1,chan)
    end;

fun numgen(n,chan) =
    let val v = CML.recv(chan)
    in if v = 0 then ()
       else CML.send(chan,n);
            numgen(n+1,chan)
    end;

fun threadcomm(n) =
    let val chan = CML.channel();
    in let val id = CML.spawn(fn()=>reader(n,chan))
       in
           numgen(n,chan);
           CML.sync(CML.joinEvt(id))
       end
    end;
```

C.8.3 Fibonacci mit Threads

```
fun fibthreadaux(n,chan) =
    CML.send(chan,
             if 2<n then
                 fibthread(n-2)+fibthread(n-1)
             else 1)

and fibthread(n) =
    let val chan = CML.channel()
    in  CML.spawn(fn()=>fibthreadaux (n,chan));
        CML.recv(chan)
    end;
```

Literaturverzeichnis

[AC91] APPLEGATE, D. und W. COOK: *A computational study of the job-shop scheduling problem*. Operations Research Society of America, Journal on Computing, 3(2):149–156, 1991.

[AFH95] AXLING, T., L. FAHLEN und S. HARIDI: *Virtual Reality Programming in Oz*. In: COCHARD, JEAN-LUC (Herausgeber): *1^{st} International Workshop on Oz Programming*, Seiten 17–24, CP 592, CH-1920 Martigny, Switzerland, 29 November–1 December 1995. IDIAP.

[App92] APPEL, ANDREW W.: *Compiling with Continuations*. Cambridge University Press, 1992.

[Arm97] ARMSTRONG, JOE: *The development of Erlang*. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Seiten 196–203. ACM Press, New York, 1997.

[AVA86] ALFRED V. AHO, RAVI SETHI, JEFFREY D. ULLMAN: *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AVWW96] ARMSTRONG, J., R. VIRDING, C. WIKSTRÖM und M. WILLIAMS: *Concurrent Programming in Erlang*. Prentice Hall, 2. Auflage, 1996.

[Bak78] BAKER, HENRY G.: *List Processing in Real Time on a Serial Computer*. Communications of the ACM, Seiten 280–294, April 1978.

[BD99] BIGOT, PETER A. und SAUMYA K. DEBRAY: *Return value placement and tail call optimization in high level languages*. The Journal of Logic Programming, 38(1):1–29, Januar 1999.

[Bel73] BELL, J. R.: *Threaded Code*. Communications of the ACM, 16:370–372, 1973.

[Bra95] BRAND, PER: *A Decision Graph Algorithm for CCP languages*. In: STERLING, LEON (Herausgeber): *Proceedings of the 1995 International Conference on Logic Programming*, Seiten 433–448, Kanagawa, Japan, Juni 1995. The MIT Press.

[But97] BUTENHOF, DAVID R.: *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Car83] CARDELLI, LUCA: *The functional abstract machine*. Polymorphism Newsletter, 1(1), 1983.

[Car91] CARLSSON, MATS: *The SICStus Emulator*. Technischer BerichtT91:15, SICS – Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1991.

[CD95] CODOGNET, PHILIPPE und DANIEL DIAZ: `wamcc`*: Compiling Prolog to C*. In: STERLING, LEON (Herausgeber): *Proceedings of the 1995 International Conference on Logic Programming*, Seiten 317–331, Kanagawa, Japan, 1995. The MIT Press.

[CFS94] CHIKAYAMA, TAKASHI, TETSURO FUJISE und DAIGO SEKITA: *A Portable and Efficient Implementation of KL1*. In: HERMENEGILDO, MANUEL und JAAN PENJAM (Herausgeber): *Programming Language Implementation and Logic Programming:* 6^{th} *International Symposium*, Seiten 25–39, Madrid, Spain, September 1994.

[CG81] CLARK, KEITH L. und STEVE GREGORY: *A Relational Language for Parallel Programming*. In: *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, Seiten 171–178. ACM Press, New York, 1981.

[CH96] CARLSSON, C. und O. HAGSAND: *DIVE – A platform for multi-user virtual environments*. Computer and Graphics, 17(6), 1996.

[Chi92] CHIKAYAMA, TAKASHI: *Operating System PIMOS and Kernel Language KL1*. In: *Proceedings of the International Conference on Fifth Generation Computer Systems*, Seiten 73–88, Tokyo, Japan, Juni 1992.

[CKC83] COLMERAUER, A., H. KANOUI und M. VAN CANEGHEM: *Prolog, Theoretical Principles and Current Trends*. Technology and Science of Informatics, 2(4):255–292, 1983.

[Con89] CONLON, T: *Programming in PARLOG*. International Series in Logic Programming. Wokingham, England - Reading, Massachusetts - Menlo Park, California: Addison-Wesley, 1989. ISBN 0-201-17450-2.

[CWY91] COSTA, VITOR SANTOS, DAVID H. D. WARREN und RONG YANG: *Andorra-I: A Parallel Prolog System that transparently exploits both And- and Or-parallelism*. In: *Proc. 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, Seiten 83–93, August 1991.

[DAK89] DAVID A. KRANZ, ROBERT H. HALSTEAD JR., ERIC MOHR: *Mul-T: A High-Performance Parallel Lisp*. In: *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Seiten 81–90, 1989.

[Deb93] DEBRAY, SAUMYA K.: *QD-Janus: a Sequential Implementation of Janus in Prolog*. SOFTWARE-PRACTICE AND EXPERIENCE, 23(12):1337–1360, Dezember 1993.

[Deu84] DEUTSCH, L. PETER: *Efficient Implementation of the Smalltalk-80 System*. In: *11th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Januar 1984.

[Dew75] DEWAR, R. B. K.: *Indirect Threaded Code*. Communications of the ACM, 18(6):330–331, Juni 1975.

[DH90] DEBRAY, SAUMYA und MANUEL HERMENEGILDO (Herausgeber): *Proceedings of the 1990 North American Conference on Logic Programming*, Austin, 1990. ALP, MIT Press.

[DMN67] DAHL, O.-J., B. MYHRHAUG und U. NYGAARD: *SIMULA-67: Common Base Language*. Norwegian Computing Center, Oslo, 1967.

[DVS+88] DINCBAS, M., P. VAN HENTENRYCK, H. SIMONIS, A. AGGOUN, T. GRAF und F. BERTHIER: *The Constraint Logic Programming Language CHIP*. In: *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, Seiten 693–702, Tokyo, Japan, Dezember 1988.

[Fra97] FRANZ, M.: *Run-Time Code Generation as a Central System Service*. In: *The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Seiten 112–117, Mai 1997.

[FT89] FOSTER, IAN und STEPHEN TAYLOR: *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Gab85] GABRIEL, RICHARD P.: *Performance and Evaluation of Lisp-Systems. 2nd Printing 1986*. Computer System Series : Research Reports and Notes. MIT Press, Cambridge, Massachusetts;London, England, 1985.

[GDBD92] GUDEMAN, DAVID, KOENRAAD DE BOSSCHERE und SAUMYA K. DEBRAY: *jc: An Efficient and Portable Sequential Implementation of Janus*. In: APT, KRZYSZTOF (Herausgeber): *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Seiten 399–413, Washington, USA, 1992. The MIT Press.

[GJ90] GUPTA, GOPAL und BHARAT JAYARAMAN: *On Criteria for Or-Parallel Execution Models of Logic Programs*. In: DEBRAY, SAUMYA und MANUEL HERMENEGILDO [DH90], Seiten 737–756.

[GJS97] GOSLING, J., B. JOY und G. STEELE: *The Java language specification*. Addison-Wesley, 1997.

[Gra93] GRANLUND, TORBJÖRN: *GNU MP: The GNU Multiple Precision Arithmetic Library*, 1.3.2 Auflage, May 1993.

[Gre87] GREGORY, STEVE: *Parallel Logic Programming in Parlog. The Language and its Implementation*. International Series in Logic Programming. Addison-Wesley, 1987.

[Gud93] GUDEMAN, DAVID: *Representing Type Information in Dynamically Typed Languages*. Technischer BerichtTR 93-27, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, Oktober 1993.

[Hal85] HALSTAEDT, ROBERT, H.: *Multilisp: A Language for Concurrent Symbolic Computation*. ACM Transactions on Programming Languages and Systems, 7(4):501–538, Oktober 1985.

[Har90] HARIDI, S.: *A Logic Programming Language Based on the Andorra Model*. New Generation Computing, 7:109–125, 1990.

[Hau94] HAUSMAN, BOGUMIL: *Turbo Erlang: Approaching the Speed of C*. In: TICK, EVAN und GIANCARLO SUCCI (Herausgeber): *Implementations of Logic Programming Systems*, Seiten 119–135. Kluwer Academic Publishers, 1994.

[Hay94] HAYGOOD, RALPH CLARKE: *Native Code compilation in SICStus Prolog*. In: HENTENRYCK, PASCAL VAN (Herausgeber): *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, Seiten 190–204, Massachusetts Institute of Technology, 1994. The MIT Press.

[HCS95] HENDERSON, F.J., T.C. CONWAY und Z. SOMOGYI: *Compiling logic programs to C using GNU C as a portable assembler*. In: *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, Seiten 1–15, 1995.

[Hen97] HENZ, MARTIN: *Objects in Oz*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 1997.

[HFA+96] HARTEL, P., M. FEELEY, M. ALT, L. AUGUSTSSON, P. BAUMANN, M. BEEMSTER, E. CHAILLOUX, C. H. FLOOD, W. GRIESKAMP, J. H. G. VAN GRONINGEN, K. HAMMOND, B. HAUSMAN, M. Y. IVORY, R. E. JONES, J. KAMPERMAN, P. LEE, X. LEROY, R. D. LINS, S. LOOSEMORE, N. RÖJEMO, M. SERRANO, J.-P. TALPIN, J. THACKRAY, S. THOMAS, P. WALTERS, P. WEIS und P. WENTWORTH: *Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark*. J. functional programming, 6(4), 1996.

[HJP92] HARIDI, SEIF, SVERKER JANSON und CATUSCIA PALAMIDESSI: *Structural operational semantics for AKL*. Future Generation Computer Systems, Seiten 409–421, 1992.

[HLZ96] HENZ, MARTIN, STEFAN LAUER und DETLEV ZIMMERMANN: *COMPOzE — Intention-based Music Composition through Constraint Programming*. In: *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, Seiten 118–121, Toulouse, France, November16–19 1996. IEEE Computer Society Press.

[HMSW97] HENZ, MARTIN, MARTIN MÜLLER, CHRISTIAN SCHULTE und JÖRG WÜRTZ: *The Oz Standard Modules*. DFKI Oz Documentation Series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[HS89] HOURI, A. und E. SHAPIRO: *A Sequential Abstract Machine for Flat Concurrent Prolog*. Journal of Logic Programming, 7:85–123, 1989.

[HU94] HÖLZLE, URS und DAVID UNGAR: *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*. In: *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Juni 1994.

[HVBS98] HARIDI, SEIF, PETER VAN ROY, PER BRAND und CHRISTIAN SCHULTE: *Programming Languages for Distributed Applications*. New Generation Computing, 16(3):223–261, 1998.

[HVS97] HARIDI, SEIF, PETER VAN ROY und GERT SMOLKA: *An Overview of the Design of Distributed Oz*. In: *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, Seiten 176–187, Maui, Hawaii, USA, Juli 1997. ACM Press.

[HW96] HENZ, MARTIN und JÖRG WÜRTZ: *Using Oz for College Time Tabling*. In: E.K.BURKE und P.ROSS (Herausgeber): *The Practice and Theory of Automated Time Tabling: The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Time Tabling, Edinburgh 1995*, Lecture Notes in Computer Science, vol. 1153, Seiten 162–177, 1996.

[Iba95] IBACH, KAI: *OzFun: A Functional Language for Mixed Eager and Lazy Programming*. Diplomarbeit, Universität des Saarlandes, Fachbereich Informatik, Oktober 1995.

[Int90] *i486 Microprocessor. Programmer's Reference Manual*. Intel, Santa Clara, California, 1990.

[Jai91] JAIN, RAJ: *The Art of Computer System Performance Analysis*. Wiley, New York;Chichester;Brisbane, 1991.

[Jan94] JANSON, SVERKER: *AKL — A Multiparadigm Programming Language*. Doktorarbeit, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994.

[JH91] JANSON, SVERKER und SEIF HARIDI: *Programming Paradigms of the Andorra Kernel Language*. In: SARASWAT, VIJAY und KAZUNORI UEDA (Herausgeber): *Logic Programming, Proceedings of the 1991 International Symposium*, Seiten 167–186, San Diego, USA, 1991. The MIT Press.

[JL87] JAFFAR, J. und J.-L. LASSEZ: *Constraint logic programming*. In: 14^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 111–119. ACM, ACM Press, New York, 1987.

[JLAW92] J. L. ARMSTRONG, B. O. DÄCKER, S. R. VIRDING und M. C. WILLIAMS: *Implementing a functional language for highly parallel real time applications*. In: *SETSS 92*, März 1992.

[KFM94] K. FISCHER, N. KUHN und J. P. MÜLLER: *Distributed, knowledge-based, reactive scheduling in the transportation domain*. In: *10th IEEE Conference on Artificial Intelligence and Applications*, New York, 1994.

[KFP95] K. FISCHER, J. P. MÜLLER und M. PISCHEL: *A Model for Cooperative Transportation Scheduling*. In: *Proceedings of the 1st International Conference on Multiagent Systems (ICMAS'95)*, San Francisco, 1995.

[KH92] KANE, GERRY und JOE HEINRICH: *MIPS RISC Architecture*. Prentice Hall, 1992.

[Kis97] KISTLER, T.: *Dynamic Runtime Optimization*. In: MÖSSENBÖCK, H. (Herausgeber): *Joint Modular Languages Conference, JMLC'97*, Nummer 1204 in *LNCS*, Seiten 53–66, Linz, März 1997. Springer-Verlag.

[Kli81] KLINT, PAUL: *Interpretation Techniques*. Software – Practice and Experience, 11(9):963–973, September 1981.

[Kor96] KORNSTAEDT, LEIF: *Definition und Implementierung eines Front-End-Generators für Oz*. Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern and Programming Systems Lab, Universität des Saarlandes, September 1996.

[Kor97] KORNSTAEDT, LEIF: *Gump: A Front-End Generator for Oz*. Programming Systems Lab, Universität des Saarlandes, Postfach 15 11 50, D-66041 Saarbrücken, Januar 1997.

[KS88] KLINGER, SHMUEL und EHUD SHAPIRO: *A Decision Tree Compilation Algorithm for FCP (I , : , ?)*. In: KOWALSKI, ROBERT A. und KENNETH A. BOWEN (Herausgeber): *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seiten 1315–1336, Seatle, 1988. ALP, IEEE, The MIT Press.

[KS90] KLIGER, SHMUEL und EHUD SHAPIRO: *From Decision Trees to Decision Graphs*. In: DEBRAY, SAUMYA und MANUEL HERMENEGILDO (Herausgeber): *North American Conference on Logic Programming*, Seiten 97–116, Austin, TX, USA, 1990. The MIT Press.

[Lan64] LANDIN, PETER J.: *The Mechanical Evaluation of Expressions*. Computer Journal, Seiten 308–320, 1964.

[Lea97] LEA, DOUG: *Concurrent Programming in Java - Design Principles and Patterns*. ADDISON-WESLEY, 1997.

[Ler97] LEROY, XAVIER: *The effectiveness of type-based unboxing*. In: *Workshop „Types in Compilation"*, Amsterdam, Juni 1997.

[LH83] LIEBERMAN, HENRY und CARL HEWITT: *A Real-Time Garbage Collector Based on the Lifetimes of Objects*. Communications of the ACM, 26(6):419–429, Juni 1983.

[LMB92] LEVINE, JOHN R., TONY MASON und DOUG BROWN: *lex & yacc*. O'Reilly & Associates, 2nd Auflage, 1992.

[Lor98] LORENZ, BENJAMIN: *Ein Debugger für Oz*. Diplomarbeit, Programming Systems Lab, Universität des Saarlandes, 1998.

[LY96] LINDHOLM, TIM und FRANK YELLIN: *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[MA95] MONTELIUS, JOHAN und KHAYRI A. M. ALI: *An And/Or-Parallel Implementation of AKL*. New Generation Computing, 13–14, August 1995.

[Mah87] MAHER, MICHAEL J.: *Logic semantics for a class of committed-choice programs*. In: LASSEZ, JEAN-LOUIS (Herausgeber): *Logic Programming, Proceedings of the Fourth International Conference*, Seiten 858–876, Cambridge, MA, 1987. The MIT Press.

[Mat97] MATTHEWS, DAVID: *Concurrency in Poly/ML*. In: *ML With Concurrency* [Nie97].

[Meh99] MEHL, MICHAEL: *The Oz Virtual Machine*. Doktorarbeit, Technische Fakultät der Universität des Saarlandes, 1999. In Vorbereitung, vorläufiger Titel.

[Mei91] MEIER, MICHA: *Recursion vs. Iteration in Prolog*. In: FURUKAWA, KOICHI (Herausgeber): *Proceedings of the Eighth International Conference on Logic Programming*, Seiten 157–169, Paris, France, 1991. The MIT Press.

[MMP+97] MEHL, MICHAEL, TOBIAS MÜLLER, KONSTANTIN POPOV, RALF SCHEIDHAUER und CHRISTIAN SCHULTE: *DFKI Oz User's Manual*. DFKI Oz Documentation Series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[Mon97] MONTELIUS, JOHAN: *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. Dissertation, SICS – Swedish Institute of Computer Science, and Uppsala University, SICS, Box 1263, SE-164 29 Kista, Sweden, 1997. SICS Dissertation Series 25.

[Moz98] *The Mozart Programming System*. `http://www.ps.uni-sb.de/mozart/`, 1998.

[MSS91] MÜLLER-SCHLOER, CHRISTIAN und ERNST SCHMITTER (Herausgeber): *RISC-Workstation-Architekturen. Prozessoren, Systeme und Produkte*. Springer-Verlag, Berlin; Heidelberg; New York, 1991.

[MSS95] MEHL, MICHAEL, RALF SCHEIDHAUER und CHRISTIAN SCHULTE: *An Abstract Machine for Oz*. In: HERMENEGILDO, MANUEL und S. DOAITSE SWIERSTRA (Herausgeber): *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95*, Lecture Notes in Computer Science, vol. 982, Seiten 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[MTHM97] MILNER, ROBIN, MADS TOFTE, ROBERT HARPER und DAVID MACQUEEN: *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[Nie97] NIELSON, FLEMMING: *ML With Concurrency*. Springer-Verlag, 1997.

[O'K90] O'KEEFE, RICHARD A.: *The Craft of Prolog*. MIT Press, 1990.

[Oz95] *The DFKI Oz Programming System*. `http://www.ps.uni-sb.de/oz1/`, 1995.

[Oz97] *The DFKI Oz Programming System (version 2)*. `http://www.ps.uni-sb.de/oz2/`, 1997.

[Pau96] PAULSON, LAURENCE C.: *ML for the working programmer*. Cambridge University Press, 1996.

[PJ87] PEYTON-JONES, SIMON L.: *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[Pop95] POPOV, KONSTANTIN: *An Exercise in Concurrent Object-Oriented Programming: The Oz Browser*. In: *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 29 November–1 December 1995.

[Pop97] POPOV, K.: *A Parallel Abstract Machine for the Thread-based Concurrent Language Oz*. In: CASTRO DUTRA, INÊS DE, VÍTOR SANTOS COSTA, FERNANDO SILVA, ENRICO PONTELLI und GOPAL GUPTA (Herausgeber): *Workshop On Parallism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[PR97] PANANGADEN, PRAKASH und JOHN REPPY: *The Essence of Concurrent ML*. In: *ML With Concurrency* [Nie97].

[RD92] ROY, PETER VAN und ALVIN M. DESPAIN: *High-Performance Logic Programming with the Aquarius Prolog Compiler*. COMPUTER, Januar 1992.

[Rep91] REPPY, JOHN H.: *CML: A Higher-order Concurrent Language*. In: *SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991. (revised 1993).

[Rep92] REPPY, JOHN HAMILTON: *Higher-Order Concurrency*. Doktorarbeit, Department of Computer Science, Cornell University, Ithaca, New York 14853-7501, 1992. availabe as Technical Report 92-1285.

[RMS96] ROY, PETER VAN, MICHAEL MEHL und RALF SCHEIDHAUER: *Integrating Efficient Records into Concurrent Constraint Programming*. In: *International Symposium on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, September 1996. Springer-Verlag.

[RV97] RÉMY, DIDIER und JÉRÔME VOUILLON: *Objective ML: A simple object-oriented extension of ML*. In: *The 24th Symposium on Principles of Programming Languages*, Seiten 40–53, Paris, France, Januar 1997.

[SA94] SHAO, ZHONG und ANDREW W. APPEL: *Space-Efficient Closure Representations*. In: *Conference on Lisp and Functional programming*, June 94.

[Sah91] SAHLIN, DAN: *An Automatic Partial Evaluator for Full Prolog*. Doktorarbeit, Swedish Institute of Computer Science, Kista, Sweden, März 1991.

[Sar89] SARASWAT, VIJAY A.: *Concurrent Constraint Programming Languages*. Doktorarbeit, School of Comp. Sc., Carnegie-Mellon University, Pittsburgh, CA, 1989.

[Sar93] SARASWAT, VIJAY A.: *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[Sch97] SCHULTE, CHRISTIAN: *Oz Explorer: A Visual Constraint Programming Tool*. In: NAISH, LEE (Herausgeber): *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, 8-11 Juli 1997. The MIT Press.

[Sch99] SCHULTE, CHRISTIAN: *Constraint Inference Engines*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 1999. In Vorbereitung, vorläufiger Titel.

[Sha83] SHAPIRO, EHUD: *A Subset of Concurrent Prolog and its Interpreter*. Technischer Bericht003, Institute for New Generation Computer Technology, Tokyo, 1983.

[Sha87] SHAPIRO, EHUD (Herausgeber): *Concurrent Prolog. Collected Papers. Volume 1 and 2*. Cambridge, Massachusetts - London: MIT Press, 1987.

[SKL90] SARASWAT, VIJAY A., KEN KAHN und JACOB LEVY: *Janus: A step towards distributed constraint programming*. In: DEBRAY, SAUMYA und MANUEL HERMENEGILDO [DH90], Seiten 431–446.

[Smo95a] SMOLKA, GERT: *The Definition of Kernel Oz*. In: PODELSKI, ANDREAS (Herausgeber): *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, Seiten 251–292. Springer-Verlag, 1995.

[Smo95b] SMOLKA, GERT: *The Oz Programming Model*. In: LEEUWEN, JAN VAN (Herausgeber): *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, Seiten 324–343. Springer-Verlag, Berlin, 1995.

[Smo97] SMOLKA, GERT: *An Oz Primer*. DFKI Oz Documentation Series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[Smo98] SMOLKA, GERT: *Concurrent Constraint Programming Based on Functional Programming (Extended Abstract)*. In: *Proceedings of the 1998 European Joint Conference on Theory and Practice of Software (ETAPS)*. Springer-Verlag, 1998.

[SPA92] *The SPARC Architecture Manual. Version 8*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[SR98] SMOLKA, GERT und ANDREAS ROSSBERG: *Object-oriented Programming Based on Functional Programming with State*. Juli 1998.

[SS96] SCHMEIER, SVEN und ACHIM SCHUPETA: *PASHA II - Personal Assistant for Scheduling Appointments*. In: *Proc. 1st Int. Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1996.

[ST94] SMOLKA, GERT und RALF TREINEN: *Records for Logic Programming*. Journal of Logic Programming, 18(3):229–258, April 1994.

[SW98] SCHILD, K. und J. WÜRTZ: *Off-Line Scheduling of a Real-Time System*. In: *Proceedings of the 1998 ACM Symposium on Applied Computing, SAC98*, Seiten 29–38, Atlanta, Georgia, 1998.

[Tan92] TANENBAUM, ANDREW S.: *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[TCS95] THOMAS CONWAY, FERGUS HENDERSON und ZOLTAN SOMOGYI: *Code generation for Mercury*. In: *Proceedings of the 1995 International Symposium on Logic Programming*, Seiten 242–256, Portland, Oregon, Dezember 1995.

[TLK97] THOMSEN, BENT, LONE LETH und TSUNG-MIN KUO: *FACILE – From Toy to Tool*. In: *ML With Concurrency* [Nie97].

[UC90] UEDA, KAZUNORI und TAKASHI CHIKAYAMA: *Design of the kernel language for the parallel inference machine*. The Computer Journal, 33(6), 1990.

[Ued85] UEDA, K.: *Guarded Horn Clauses*. Technischer BerichtTR-103, ICOT, 1985. see also [Sha87].

[VHB+97] VAN ROY, PETER, SEIF HARIDI, PER BRAND, GERT SMOLKA, MICHAEL MEHL und RALF SCHEIDHAUER: *Mobile Objects in Distributed Oz*. ACM Transactions on Programming Languages and Systems, 19(5):804–851, September 1997.

[VR90] VAN ROY, PETER LODEWIJK: *Can Logic Programming Execute as Fast as Imperative Programming*. Doktorarbeit, University of California, Computer Science Division (EECS), Berkeley, California 94720, 1990. Report No. UCB/CSD 90/6000.

[Wal96] WALSER, JOACHIM: *Feasible Cellular Frequency Assignment Using Constraint Programming Abstractions*. In: *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, Massachusetts, USA, August 1996.

[WAMR97] WILHELM, REINHARD, MARTIN ALT, FLORIAN MARTIN und MARTIN RABER: *Parallel Implementation of Functional Languages*. In: DAM, MADS (Herausgeber): *5th LOMAPS Workshop, Analysis and Verification of Multiple-Agent Languages*, Band 1192 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, June 1997.

[War77] WARREN, D.: *Compiling Predicate Logic Programs*. DAI Research Reports 39 and 40, University of Edinburgh, 1977.

[War83] WARREN, D.: *An Abstract Prolog Instruction Set*. Technical Report 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, 1983.

[War87] WARREN, DAVID H.D.: *The Andorra Principle*. Presented at the Gigalips workshop, Stockholm, 1987.

[Wik96] WIKSTRÖM, CLAES: *Implementing Distributed Real-time control systems in a functional language*. In: *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1996.

[WM97] WILHELM, REINHARD und DIETER MAURER: *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite Auflage*. Springer-Verlag, 1997.

[Wür98] WÜRTZ, JÖRG: *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, Januar 1998.

[Wür96] WÜRTZ, J.: *Oz Scheduler: A Workbench for Scheduling Problems*. In: RADLE, M.G. (Herausgeber): *Eighth International Conference on Tools with Artificial Intelligence*, Seiten 149–156, Toulouse, France, 1996. IEEE, IEEE Computer Society Press.

Index

Symbole
∼, 18
`::`, 14

A
Abschluß, 19
ACL, *siehe* Allegro Common Lisp
Agents, 216
AKL, *siehe* Andorra Kernel Language
Allegro Common Lisp, 188
`allocate`, 62
Andorra Kernel Language, 215
Andorra-Prinzip, 217
`ap`, *siehe* Argumentzeiger
`append`, 29
`appendrec`, 33
`apply`, 61
Argumentzeiger, 89, 124
Arität
    eines Tupels, 16
Atom, 92
Auftrag, 52
Ausnahmebehandlung, 32

B
BEAM, 216
Beta-Reduktion, 104
Binden
    einer Variable, 25
`branch`, 70
Browser, 233
Builtin, *siehe* Funktion,nativ

C
C++, 127
`cachedApply`, 102
`case`, 22, 28
`caseStart`, 124
cc, *siehe* concurrent constraint
closure, *siehe* Abschluß
CML, *siehe* Concurrent ML
Code
    nativer, 41
Codebereich, 125
Codespezialisierung, 81
    dynamisch, 83
    statisch, 82
Compiler, *siehe* Übersetzer
concurrent constraint, 217
Concurrent ML, 214
`consp`, 57
Constraintmodell, 11
CPS, 143
Currying, 20
`curUVARPrototye`, 163
`CVAR`, 161

D
`deallocate`, 62
Dereferenzieren, 45
Determiniertheit, 27
DFKI Oz, 1
Dissubsumption, 28

E
EGCS, 189
Emulator, 4, 41
Endrekursion, 99
environment trimming, 117
Erlang, 188, 216
Evaluierung, 19
    lazy, 119
`exchange`, 58
`exchange`, 24
Explorer, 234
Export
    von Variablen, 74
`export`, 75
Exporttabelle, 75

F
FACILE, 214

Fairneß, 31, 53, 171
false, 14
falsep, 57
fastApply, 101
Fibonacci, 193
freelist, 125
Freispeicherverwaltung, 140
fun, 60
funCopy, 87
Funktion
    nativ, 177
Funktor, 86
Future, 213

G
G, 124
GCL, *siehe* GNU Common Lisp
GCTAG, 148
getCon, 70
getInt, 70
getList, 92
getTuple, 70
getVarArg, 160
GNU Common Lisp, 188
Graphenmodell, 11
guard, *siehe* Wächter
guardEnd, 70
guardStart, 70
Gump, 234

H
Halde, 139
heapEnd, 125
heapSave, 124
heapTop, 125

I
import, 75
Indexierung, 97
Inline-Caching, 101
Instruktion, 42
Interpreter, 128

J
JAM, 216
Janus, 218
Java, 188
JDK, 188

K
Kaffe, 188
Keller, 108
Kellervariablen, 160
KL1, 218
Knoten, 15
Komponente
    eines Tupels, 16
Konstruktoren, 16

L
L, 11
L, 124
Lader, 43, 50
Laufzeitfehler, 32
LIST, 149
Listen, 14, 91
Literal, 152
loadArg, 95

M
Mandelbrot, 199
Marke
    eines Tupels, 16
Markierte Referenzen, 145
Maschine
    reale, 4, 41
    virtuelle, 4, 41, 43
Maschinencode, 125
Maschinenprogramm, 43, 47
MLWorks, 188
mode, *siehe* Modus-Register
Modus-Register, 90, 124
Monotonie, 31
Moscow ML, 188
move, 60
moveArg, 70, 89, 90
Mozart, 1
Mul-T, 214
Multilisp, 213

N
N-Damen, 198
Naive Reverse, 195
Namen
    globale, 79
newCon, 58
newList, 91
newRef, 58
newTuple, 88

`newVar`, 58
`nil`, 14
`nilp`, 57

O
OCAML, 188
Operationen, 23
Ozcar, 234

P
Parallelität, 3
`PC`, 124
Penny, 216
Persistenz, 77
`plus`, 59
`plusConst`, 93
Poly/ML, 214
Preemption, 52
Prelude, 48, 233
Programm, 19
Programmvariable, 25
Programmzähler, 52

Q
Quicksort, 196

R
`REF`, 44, 149
Referenzketten, 109
Register, 46
    allgemeine, 46
    G-Register, 47
    globale, 47
    L-Register, 47
    lokale, 47
    V-Register, 47
    virtuelle, 47
    X-Register, 46
Registerallokation, 46, 115
Registermaschine, 107
`return`, 63
Rumpf
    eines Case, 22
`running`, 124
`runqueue`, 124

S
Scheduler, 127
Segment, 49
`setArg`, 89
`setVarArg`, 160
SICS, 1
SICStus Prolog, 188
Signalbehandlung, 52, 171
Simula, 2
SML/NJ, 188
`SP`, 124
`spawn`, 32, 59
`specApply`, 101
Spedition, 234
Speicherbereinigung, 143
Spur, 67
Stackmaschine, 107
Subsumption, 28
Suspension, 27, 31, 53
    eines Builtins, 179
Suspensionsliste, 54
`suspVars`, 125
`switch`, 98
Symbolische Ableitung, 200
Synchronisation, 3

T
Takeushi, 191
TEAM, 216
`testBool`, 96
`testConst`, 94
`testInteger`, 94
`testLess`, 97
`testList`, 95
`testTuple`, 95
Thread, 2, 30
    Zustände, 30
threaded code, 129
trail, *siehe* Spur
`trail`, 124
`true`, 14
`truep`, 57
Tupel, 16

U
Übersetzer, 43
Umgebung, 19
Unifikation, 25
`unifyConst`, 88
`unifyInteger`, 88
`unit`, 14
`unitp`, 57

Universum, 15
`UVAR`, 162

V
`vaddHTCon`, 98
`vaddHTInt`, 98
`vaddHTTuple`, 98
value, *siehe* Wert
Variable
    globale, 66
    logische, 25
    lokale, 66
    permanent, 116
    Programm-, 25
    temporär, 116
Verteilung, 78
`vmakeHT`, 98
`vnewCon`, 48
`vnewInt`, 48
`vnewVar`, 48

W
Wächter, 3, 22
    flach, 67
    linear, 22
    tief, 67, 115
`wait`, 29
WAM, 215
Wert, 15

Z
Zellen, 16, 23
Zyklen, 25