

Reinhard Wilhelm · Helmut Seidl

Übersetzerbau

Virtuelle Maschinen

Mit 120 Abbildungen

 Springer

Reinhard Wilhelm

Universität des Saarlandes
66123 Saarbrücken
wilhelm@cs.uni-sb.de

Helmut Seidl

Technische Universität München
Fakultät für Informatik
Boltzmannstraße 3
85748 Garching
seidl@in.tum.de

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen
Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
<http://dnb.d-nb.de> abrufbar.

Das vorliegende Buch ist als Neuauflage aus dem Buch Wilhelm, R.; Maurer, D. *Übersetzerbau: Theorie, Konstruktion, Generierung* hervorgegangen, das in der 1. Auflage (ISBN 3-540-55704-0) und der 2. Auflage (ISBN 3-540-61692-6) im Springer-Verlag erschien.

ISSN 1614-5216

ISBN 978-3-540-49596-3 Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

© Springer-Verlag Berlin Heidelberg 2007

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Text und Abbildungen wurden mit größter Sorgfalt erarbeitet. Verlag und Autor können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3100 YL – 5 4 3 2 1 0

Für Margret, Hannah, Eva, Barbara

R.W.

Für Kerstin und Anna

H.S.

Vorwort

Übersetzer für höhere Programmiersprachen sind große komplexe Softwaresysteme. Sie haben aber einige besondere Eigenschaften, die sie vor den meisten anderen Softwaresystemen auszeichnen.

Ihre Funktionalität ist (fast) wohldefiniert. Idealerweise existieren vollständige formale oder zumindest präzise Beschreibungen der Quellsprache und der Zielsprache. Häufig gibt es dazu noch Beschreibungen von Schnittstellen zum Betriebssystem, zum Programmiersystem und zu Programmierumgebungen, zu anderen Übersetzern und zu Programmbibliotheken.

Die Übersetzungsaufgabe lässt sich auf natürliche Weise in Teilaufgaben zerlegen. Diese Zerlegung ergibt eine modulare Struktur, welche übrigens auch eine kanonische Struktur der üblichen Übersetzerbaubücher induziert.

In den fünfziger Jahren wurde bereits erkannt, dass die Implementierung von Anwendungssystemen direkt in der Maschinensprache sowohl mühsam als auch fehleranfällig ist und zu Programmen führt, die genauso schnell veralten wie die Rechner, für die sie entwickelt wurden. Mit der Erfindung höherer maschinenunabhängiger Programmiersprachen ergab sich aber auch sofort die Notwendigkeit, Übersetzer bereitzustellen, die in der Lage sind, Programme dieser höheren Programmiersprachen in Maschinensprache zu übersetzen.

Aufgrund dieser grundlegenden Herausforderung sind deshalb seit den fünfziger Jahren die verschiedenen Teilaufgaben der Übersetzung Gegenstand intensiver Forschung. Für die Teilaufgabe der syntaktischen Analyse von Programmen wurden etwa Konzepte aus dem Bereich der formalen Sprachen wie endliche Automaten und kontextfreie Grammatiken übernommen und im Hinblick auf die gegebene Anwendung weiterentwickelt. Die theoretische Durchdringung der Problemstellung war dabei so erfolgreich, dass die Realisierung der für syntaktische Analyse benötigten Komponenten (fast) vollständig automatisiert werden konnte: anstatt „zu Fuß“ implementiert zu werden, werden diese Komponenten heute weitgehend aus Spezifikationen, in diesem Fall kontextfreie Grammatiken, generiert. Solche Generierungsverfahren werden auch für andere Komponenten eines Übersetzers angestrebt und sind zum Teil bereits realisiert.

Das vorliegende Buch strebt nicht an, ein Kochbuch für Übersetzer zu sein. Man wird hier darum keine Rezepte finden der Art: „Um einen Übersetzer von der Quellsprache X in die Maschinensprache Y zu konstruieren, nehme man ...“. Unsere Darstellung reflektiert dagegen die oben aufgezählten Besonderheiten des Übersetzerbaus, insbesondere das Vorhandensein präziser Aufgabenstellungen und das Bestreben, diese genau zu verstehen und angemessene theoretische Konzepte zu ihrer systematischen Behandlung bereitzustellen. Idealerweise können solche Konzepte die Grundlage automatischer Generierungsverfahren bilden.

Dieses Buch ist für Studierende der Informatik bestimmt. Die Kenntnis zumindest einer imperativen Programmiersprache wird vorausgesetzt. Für die Kapitel zur Übersetzung funktionaler und logischer Programmiersprachen ist es sicher nützlich, eine moderne funktionale Sprache und Prolog zumindest in den Grundzügen zu kennen. Andererseits können diese Kapitel auch zu einem vertieften Verständnis dieser Programmiersprachen beitragen.

Aufbau des Buches

Für die neue Auflage des Buchs, Wilhelm/Maurer: *Übersetzerbau*, entschieden wir uns, den Inhalt auf mehrere Bände zu verteilen. Dieser erste Band beschreibt, *was* ein Übersetzer tut, also welche Korrespondenz zwischen einem Quellprogramm und einem Zielfprogramm er herstellt. Dazu wird für eine imperative, eine funktionale, eine logische und eine objektorientierte Programmiersprache je eine geeignete *virtuelle* Maschine (in den früheren Auflagen *abstrakte* Maschine genannt) angegeben und die Übersetzung von Programmen der jeweiligen Quellsprache in die Sprache der zugehörigen virtuellen Maschine genau beschrieben.

Die virtuellen Maschinen der vorherigen Ausgabe wurden vollständig überarbeitet und modernisiert mit dem Ziel, die Übersetzungsschemata zu vereinfachen und gegebenenfalls zu vervollständigen. In viel größerem Umfang als vorher wurden die verschiedenen gewählten Architekturen und Befehlssätze aneinander angeglichen, um die Gemeinsamkeiten, aber auch Unterschiedlichkeiten der Sprachkonzepte klarer herauszuarbeiten. Als vielleicht augenfälligstes, wenn auch nicht unbedingt wichtigstes Merkmal wachsen jetzt, wie ein Leser früherer Auflagen sofort erkennen wird, die Keller der virtuellen Maschinen (wie manche Träume) in den Himmel, also nicht mehr „von oben nach unten“.

Bei allen Beispiel-Programmiersprachen wurden nun Fragmente realer Programmiersprachen verwendet. Als imperative Quellsprache wurde anstelle von Pascal die Programmiersprache C gewählt – eine Entscheidung für mehr Realitätsnähe und gegen die Ästhetik, wie man beim Vergleich der entsprechenden Kapitel in dieser und früheren Auflagen feststellen kann. Als objektorientierte Sprache dient wieder eine Teilmenge von C++. Gegenüber der Darstellung der zweiten Auflage wurde jedoch auf eine detaillierte Diskussion der Mehrfachbeerbung verzichtet.

Ausgangspunkt der in diesem Buch angegebenen Übersetzungen von imperativen, funktionalen, logischen und objektorientierten Programmen ist stets eine strukturierte Interndarstellung des Quellprogramms, für welches bereits einfache Zusatz-

informationen wie etwa Gültigkeitsbereiche von Variablen oder Typinformationen berechnet wurden. Ein solches analysiertes Quellprogramm werden wir später als *dekorierte abstrakte Syntax* des Quellprogramms bezeichnen.

Im zweiten Band wird dann das *Wie* der Übersetzung beschrieben. Wir werden die Frage behandeln, wie man den Übersetzungsprozess in einzelne Phasen unterteilt, welche Aufgaben dabei die einzelnen Phasen erledigen sollen, welche Techniken man in ihnen benutzt, wie man formal beschreiben kann, was sie tun, und wie eventuell aus einer solchen Beschreibung automatisch ein Übersetzermodul erzeugt werden kann.

Danksagung

Neben den Mitstreitern der früheren Auflagen möchten wir den Studierenden danken, die immer wieder verschiedene Versionen der virtuellen Maschinen geduldig anhörten und uns wertvolle Rückmeldungen gaben. Nicht wenig zum Verständnis trug auch die Visualisierung der virtuellen Maschinen durch Peter Ziewer bei. Für subtile Einsichten in die Semantiken von C++ und Java danken wir Thomas Gawlitza und Michael Petter. Unser besonderer Dank gebührt jedoch Jörg Herter, der mehrere Versionen des Buchs sorgfältig auf Inkonsistenzen durchsah und uns auf vielerlei Fehler und Merkwürdigkeiten aufmerksam machte.

Einstweilen wünschen wir aber der geneigten Leserin und dem geneigten Leser viel Vergnügen mit dem vorliegenden Band und hoffen, dass das Buch Appetit machen möge, für die Lieblings-Programmiersprache blitzschnell einen eigenen Übersetzer zu basteln.

Saarbrücken und München, Januar 2007

Reinhard Wilhelm, Helmut Seidl

Weitere Materialien zu diesem Buch finden Sie auf der Internet-Seite:

<http://www2.informatik.tu-muenchen.de/~seidl/compiler/>

Inhaltsverzeichnis

1	Einleitung	1
1.1	Höhere Programmiersprachen	1
1.2	Implementierung von Programmiersprachen	2
1.2.1	Interpreter	2
1.2.2	Übersetzer	3
1.2.3	Reale und virtuelle Maschinen	4
1.2.4	Kombinationen von Übersetzung und Interpretation	5
1.3	Allgemeine Literaturhinweise	6
2	Imperative Programmiersprachen	7
2.1	Sprachkonzepte und ihre Übersetzung	7
2.2	Die Architektur der C-Maschine	8
2.3	Einfache Ausdrücke und Wertzuweisungen	10
2.4	Anweisungen und Anweisungsfolgen	15
2.5	Bedingte und iterative Anweisungen	16
2.6	Speicherbelegung für Variablen einfachen Typs	22
2.7	Speicherbelegung für Felder und Strukturen	23
2.8	Zeiger und dynamische Speicherbelegung	28
2.9	Funktionen	34
2.9.1	Die Speicherorganisation der C-Maschine	37
2.9.2	Der Umgang mit lokalen Variablen	39
2.9.3	Betreten und Verlassen von Funktionen	42
2.10	Übersetzung ganzer Programme	48
2.11	Aufgaben	51
2.12	Liste der Register der CMa	57
2.13	Liste der Codeerzeugungsfunktionen der CMa	57
2.14	Liste der CMa-Instruktionen	58
2.15	Literaturhinweise	58

3	Funktionale Programmiersprachen	59
3.1	Sprachtyp und einleitende Beispiele	59
3.2	Eine einfache funktionale Programmiersprache	62
3.3	Die Architektur der MaMa	66
3.4	Die Übersetzung einfacher Ausdrücke	68
3.5	Der Zugriff auf Variablen	71
3.6	<i>let</i> -Ausdrücke	75
3.7	Funktionsdefinitionen	76
3.8	Funktionsanwendungen	79
3.9	Unter- und Überversorgung mit Argumenten	82
3.10	Rekursive Variablendefinitionen	86
3.11	Abschlüsse und ihre Auswertung	89
3.12	Optimierungen I: Globale Variablen	92
3.13	Optimierungen II: Abschlüsse	93
3.14	Die Übersetzung eines Programmausdrucks	94
3.15	Strukturierte Daten	95
3.15.1	Tupel	95
3.15.2	Listen	97
3.15.3	Abschlüsse für Tupel und Listen	100
3.16	Optimierungen III: Letzte Aufrufe	101
3.17	Aufgaben	102
3.18	Liste der Register der MaMa	106
3.19	Liste der Codeerzeugungsfunktionen der MaMa	107
3.20	Liste der MaMa-Instruktionen	107
3.21	Literaturhinweise	107
4	Logische Programmiersprachen	109
4.1	Die Sprache ProL	109
4.2	Die Architektur der WiM	112
4.3	Anlegen von Termen in der Halde	114
4.4	Die Übersetzung von Literalen	118
4.5	Unifikation	120
4.6	Klauseln	130
4.7	Die Übersetzung von Prädikaten	131
4.7.1	Backtracking	132
4.7.2	Zusammenfügung	134
4.8	Die Endbehandlung von Klauseln	135
4.9	Anfragen und Programme	137
4.10	Optimierung I: Letzte Ziele	140
4.11	Optimierung II: Verkleinerung von Kellerrahmen	143
4.12	Optimierung III: Klausel-Indizierung	144
4.13	Erweiterung: Der Cut Operator	147
4.14	Exkurs: Speicherbereinigung	149
4.15	Aufgaben	154
4.16	Liste der Register der WiM	157

4.17	Liste der Codeerzeugungsfunktionen der WiM	157
4.18	Liste der WiM-Instruktionen	157
4.19	Literaturhinweise	158
5	Objektorientierte Programmiersprachen	159
5.1	Konzepte objektorientierter Sprachen	159
5.1.1	Objekte	159
5.1.2	Objektklassen	160
5.1.3	Vererbung	161
5.1.4	Generizität	162
5.1.5	Informationskapselung	163
5.1.6	Zusammenfassung	163
5.2	Eine objektorientierte Erweiterung von C	164
5.3	Die Speicherorganisation für Objekte	165
5.4	Methodenaufrufe	168
5.5	Die Definition von Methoden	171
5.6	Die Verwendung von Konstruktoren	171
5.7	Die Definition von Konstruktoren	173
5.8	Ausblick: Mehrfache Beerbung	175
5.9	Aufgaben	177
5.10	Liste zusätzlicher Register	182
5.11	CMA-Instruktionen für Objekte	182
5.12	Literaturhinweise	182
	Literaturverzeichnis	183
	Sachverzeichnis	187

Einleitung

1.1 Höhere Programmiersprachen

Programme werden heute zumeist in sogenannten problemorientierten, höheren Programmiersprachen geschrieben. Diese Programmiersprachen abstrahieren (in verschiedenem Maße) von der Struktur und den Details der Rechner, auf denen die geschriebenen Programme ausgeführt werden sollen. Die vier wichtigsten Klassen von universell einsetzbaren Programmiersprachen sind

- die Klasse der *imperativen Sprachen* wie etwa Algol 60, Algol 68, Fortran, Cobol, Pascal, Ada, Modula-2 und C. Sie orientieren sich eng an der Struktur des sogenannten von-Neumann-Rechners, die fast allen kommerziell erwerbbaaren Rechnern zugrunde liegt und aus (aktiver) Zentraleinheit (CPU), (passivem) Speicher und einem Bus für den Verkehr zwischen Zentraleinheit und Speicher besteht.
- die Klasse der *funktionalen Sprachen* wie etwa pure Lisp, SML, OCaml und Haskell. Charakteristisch für diese Klasse ist es,
 - dass es keine Trennung zwischen der Welt der Anweisungen und der Ausdrücke gibt,
 - dass Namen nur als Bezeichner für Ausdrücke und Funktionen aber nicht für Speicherzellen dienen und
 - dass Funktionen als Argumente und Ergebnisse von Funktionen auftreten dürfen.

Das Ausführungsprinzip ist *Reduktion*; d.h. ein funktionales Programm wird ausgewertet, indem in einzelnen Schritten so lange jeweils ein (Teil-) Ausdruck durch einen äquivalenten, einfacheren ersetzt wird, bis dieser Prozess mit Erreichen der Normalform endet.

- die Klasse der *logischen Programmiersprachen* wie Prolog und seine verschiedenen Dialekte. Diese Sprachen basieren auf einer operationellen Sicht der Prädikatenlogik. Der Ausführungsmechanismus ist *Resolution*, ein Verfahren, das für das Beweisen von Implikationen in der Prädikatenlogik der ersten Stufe entwickelt wurde.

- die Klasse der *objektorientierten Programmiersprachen* wie etwa Smalltalk, C++ oder Java. Ihre Vertreter sind im Kern imperativ, verfügen gegebenenfalls über Typsysteme, die Datenabstraktion unterstützen und eine „evolutionäre“ Art der Softwareentwicklung ermöglichen.

Neben diesen Klassen gibt es noch viele Sprachen für spezielle Anwendungen:

- Hardware-Beschreibungssprachen wie etwa VHDL. Sie dienen zur Spezifikation von Rechnern und Rechnerkomponenten. Solche Spezifikationen können das funktionale Verhalten, den hierarchischen Aufbau und die geometrische Platzierung von Komponenten beschreiben.
- Kommandosprachen von Betriebssystemen. Sie besitzen als primitive Konstrukte u.a. die Aktivierung von Systemfunktionen und Benutzerprogrammen und die Möglichkeit, mehrere solcher Programme und Systemfunktionen koordiniert zusammenwirken zu lassen, Prozesse zu erzeugen und zu beenden, Ausnahmesituationen zu entdecken und zu behandeln.
- Spezifikationssprachen für Druckseiten, Graphikobjekte oder Animationen. Ein Beispiel ist hier etwa die Programmiersprache Postscript von Adobe, die im Drucker das Erscheinungsbild der jeweiligen Seiten berechnet. Im Falle von Animationen müssen nicht nur die geometrischen Dimensionen der darzustellenden Objekte beschrieben werden, sondern auch zeitliche Abläufe und möglicherweise vorgesehene Reaktionen auf Ereignisse.
- Sprachen zur Bearbeitung strukturierter Dokumente. In den letzten Jahren hat sich XML als Standardformat zur Repräsentation strukturierter Daten durchgesetzt. Die Fülle der Anwendungen und die Weite der Verbreitung im Internet führte zu einer Vielzahl weiterer Standards, angefangen von XSchema zur sehr genauen Beschreibung von Dokumenten über die XML-Transformationssprache XSLT und die XML-Anfragesprache XQuery bis hin zu Formalismen im Zusammenhang mit Web Services oder Business Processes.

1.2 Die Implementierung von Programmiersprachen

Damit Programme einer bestimmten Programmiersprache L auf einem Rechner ausgeführt werden können, muss diese Programmiersprache auf diesem Rechnertyp verfügbar gemacht, man sagt, *implementiert* werden. Dies kann auf verschiedene Weise geschehen. Man teilt die Implementierungen in interpretierende und übersetzende Verfahren ein.

1.2.1 Interpreter

Wir betrachten eine Programmiersprache L . Ein Interpreter I_L bekommt als Eingabe ein Programm p_L aus L und eine Eingabefolge e und berechnet daraus eine Ausgabefolge a . Eventuell führt die Interpretation von p_L auch zu einem Fehler. Also hat I_L die Funktionalität

$$I_L : L \times D^* \rightarrow D^* \times \{error\},$$

wenn sowohl Eingabe- wie Ausgabedaten aus einem Bereich D stammen. Die oben geschilderte Ausführung des Programms p_L mit Eingabefolge e und Ergebnisfolge a ist dann durch die Gleichung

$$I_L(p_L, e) = a$$

beschrieben.

Was ist kennzeichnend für die Arbeitsweise eines Interpreters? Er bearbeitet das Programm p_L und die Eingabe e zur gleichen Zeit. Jedem Konstrukt steht er – auch bei erneuter Ausführung – unvorbereitet gegenüber. Er nutzt keine von den Eingabedaten unabhängige Informationen aus, die er durch Inspektion des Programmtextes gewinnen könnte, etwa die Zahl der deklarierten Variablen in einem Block, einer Funktion oder einer Klausel. Diese Information könnte er für eine Speicherzuteilung mit effizientem Zugriff auf die Variablenwerte benutzen.

1.2.2 Übersetzer

Es gilt die mit der Interpretation verbundenen Ineffizienzen zu vermeiden. Dazu benutzt man ein in der Informatik häufig nützliches Prinzip, meist *Vorberechnung*, manchmal auch *partielle Auswertung* (Partial Evaluation) oder *gemischte Berechnung* (Mixed Computation) genannt.

Während der Interpreter I seine beiden Argumente, das Programm p_L und die Eingabefolge e , zur gleichen Zeit bekommt und verarbeitet, wird jetzt die Verarbeitung des Programms und der Eingabefolge auf zwei verschiedene Zeiten aufgeteilt. Erst wird das Programm p_L „vorverarbeitet“, d.h. unabhängig von Eingabedaten analysiert und in eine Form überführt, welche die effizientere Ausführung des Programms mit beliebigen Eingabefolgen erlaubt. Man nimmt dabei an, dass sich der zusätzliche Aufwand für die Vorverarbeitung des Programms bei der Ausführung auf einer oder mehreren Eingabefolgen amortisiert.

Wie sieht die Vorverarbeitung von Programmen aus? Meist besteht sie in der Übersetzung des Programms p_L der Sprache L , ab jetzt *Quellsprache* genannt, in die Maschinen- oder Assemblersprache M eines konkreten oder virtuellen Rechners. Die Zeit, zu der diese Übersetzung geschieht, heißt folglich *Übersetzungszeit*, das sich ergebende Programm p_M *Zielprogramm* zu p_L .

Bei der Übersetzung von Quellprogrammen in Zielprogramme werden insbesondere die beiden oben aufgeführten Quellen von Ineffizienz bei der Interpretation beseitigt. Jedes Programm wird einmal zur Übersetzungszeit analysiert; das erzeugte Zielprogramm – nehmen wir an, es sei ein Programm in der Maschinensprache eines realen Rechners – erfährt an Analyse nur die Decodierung des Befehlscodes durch die Befehlseinheit des Rechners. Der effiziente, teilweise direkte Zugriff auf Variablenwerte bzw. Zellen wird durch ein Speicherverwaltungsschema ermöglicht, das allen Variablen des Programms feste (Relativ-) Adressen zuordnet. Diese Adressen stehen dann auch im erzeugten Zielprogramm.

Das erzeugte Zielprogramm p_M wird zu einer auf die Übersetzungszeit folgenden Zeit, genannt *Laufzeit*, mit der Eingabefolge e ausgeführt. Natürlich verlangen

wir von der Übersetzung, dass das Zielprogramm p_M bei der Ausführung mit Eingabe e nach Möglichkeit genau die Ergebnisfolge produziert, die der Interpreter auf p_L und e liefert.

Eine Mindestanforderung für die Übersetzung ist darum die folgende: Sei p_L ein Programm, welches syntaktisch korrekt ist und außerdem den Kontextbedingungen von L genügt. Der Übersetzer erzeuge für p_L das Zielprogramm p_M . Stößt I_L bei der Ausführung von p_L mit e auf keinen Fehler und produziert die Ausgabe a , so stößt auch p_M bei der Ausführung mit Eingabefolge e auf keinen Fehler, und die Interpretation und die Ausführung von p_M auf e liefern das gleiche Ergebnis.

Fassen wir die Maschine M als einen Interpreter I_M für ihre Maschinensprache auf, so muss für solche Kombinationen (p_L, e) also gelten:

$$I_L(p_L, e) = a \quad \implies \quad I_M(p_M, e) = a$$

Dazu gibt es mehrere Fehlersituationen. Einmal kann p_L syntaktische Fehler oder Verletzungen der Kontextbedingungen in Programmteilen enthalten, die der Interpreter bei der Ausführung mit e gar nicht berührt. Dann könnte die Interpretation erfolgreich ablaufen, während der Übersetzer, der das ganze Programm analysiert, die Übersetzung in ein Zielprogramm wegen entdeckter Fehler ablehnt. Zum anderen kann I_L , obwohl p_L syntaktisch und gemäß der Kontextbedingungen korrekt ist, bei der Ausführung mit Eingabe e auf einen (Laufzeit-) Fehler stoßen. Wenn wir I_L als die Definition der Semantik von L betrachten, muss dann auch das erzeugte Zielprogramm p_M bei der Ausführung mit e auf einen Fehler stoßen.

1.2.3 Reale und virtuelle Maschinen

In der Regel wird dem Programmierer ein als *real* bezeichneter Rechner zur Verfügung stehen; reale Rechner sind in großer Vielfalt käuflich zu erwerben und zwar in Form von Hardware, d.h. Platinen bestückt mit irgendeinem Prozessor, Speicherchips, und was sonst noch erforderlich ist. Die Zielsprache der Übersetzung ist in diesem Fall durch den verwendeten Prozessortyp definiert.

Will man für eine höhere Programmiersprache Code erzeugen, wird man jedoch schnell feststellen, dass man bei der Übersetzung gerne Befehle verwenden würde, die so von einer gegebenen realen Maschine nicht bereit gestellt werden. Andererseits ändern sich die Instruktionssätze moderner Rechner so schnell, dass es nicht sinnvoll erscheint, den Compiler zu sehr auf zufälligerweise bereitgestellte Operationen festzulegen. Eine solche Festlegung könnte nämlich bedeuten, dass man nach wenigen Jahren den Compiler für die nächste Generation von Rechnern neu schreiben müsste.

Bereits bei der Implementierung des ersten Pascal-Übersetzers kam man darum auf die Idee, zuerst Code für eine leicht idealisierte *virtuelle* Maschine zu erzeugen, deren Befehle dann jeweils nur noch auf den verschiedenen konkreten Zielrechnern zu implementieren waren. Auch die Übersetzung moderner Programmiersprachen wie Prolog, Haskell oder Java basieren auf diesem Prinzip. Einerseits erleichtert dieses Vorgehen die Portierbarkeit des Übersetzers. Andererseits vereinfacht dies die

Übersetzung selbst, da man den Befehlssatz entsprechend der jeweiligen zu übersetzenden Programmiersprache geeignet wählen kann.

Neue Anwendungen im Bereich des Internet haben die Idee virtueller Maschinen seit einiger Zeit zusätzlich attraktiv gemacht. Die Portierbarkeit, die man durch eine Implementierung einer Programmiersprache auf einer virtuellen Maschine gewinnt, kann man ausnutzen, um Systeme *plattformunabhängig* zu realisieren, d.h. so, dass sie unter verschiedenen Betriebssystemen lauffähig sind. Noch etwas weiter gedacht, kann Code so sogar *mobil* gemacht und über das Internet verbreitet werden. Das ist unter anderem die Idee hinter der Programmiersprache Java.

Mit der Ausführung fremden Codes auf dem eigenen Rechner handelt man sich jedoch nicht nur Vergnügen ein, sondern setzt sich auch den Angriffen böswilliger Angreifer aus. Hier bietet eine virtuelle Maschine ebenfalls eine Lösung: da der Code nicht direkt auf der eigenen Hardware ausgeführt wird, kann man das Verhalten des auszuführenden Codes genau beobachten bzw. seine Zugriffsrechte auf die Ressourcen des Rechners entsprechend einschränken. Diese Idee nennt man auch das *Sandkasten-Prinzip* (Sand Boxing).

In diesem Buch stellen wir virtuellen Maschinen für imperative, funktionale, logische und objektorientierte Programmiersprachen vor. Insbesondere sind wir dabei natürlich an den Übersetzungsschemata interessiert, wie man für die jeweiligen konkreten Programmkonstrukte der Programmiersprache die zugehörigen Befehlsfolgen der virtuellen Maschinen konstruiert.

1.2.4 Kombinationen von Übersetzung und Interpretation

Verschiedene Kombinationen von Übersetzung und Interpretation sind möglich. Werden Quellprogramme in die Maschinensprache einer virtuellen Maschine übersetzt, so wird diese meist durch einen Interpreter, einem Stück Software interpretiert. Allerdings kann man auch die Sprache der virtuellen Maschine weiter übersetzen, z.B. in die Sprache eines realen Rechners, heute *nativer Code* genannt. Interessante Unterschiede ergeben sich, je nachdem zu welcher Zeit und auf welcher Maschine dieser zweite Übersetzungsschritt erfolgt. Er kann nämlich auch zur Ausführungszeit erfolgen und wird dann *just-in-time Übersetzung (JIT)* genannt. Meist werden die beiden Schritte auf zwei verschiedenen Rechnern ausgeführt. Der erste Schritt passiert auf dem Rechner des Programmierers. Hier wird ein portabler Zwischencode in der Sprache einer virtuellen Maschine erzeugt. Dieser wird eventuell auf einen anderen Rechner heruntergeladen. Der zweite Übersetzungsschritt wird auf diesem Rechner durchgeführt und erzeugt möglichst effizienten nativen Code für diesen Rechner. Es werden also Übersetzungszeit und Ausführungszeit kombiniert; der JIT-Übersetzer wird zur Ausführungszeit für ein Stück des Programms, z.B. eine Funktion aufgerufen, meist zu dem Zeitpunkt, wenn diese Funktion gebraucht wird.

Das Ziel der JIT-Übersetzung ist die Kombination von Effizienz und Portierbarkeit. Der erste Übersetzungsschritt, der im Allgemeinen sehr aufwendig ist, wird nur einmal gemacht. Der zweite Schritt, die Übersetzung des Zwischencodes, ist einfacher. Der erzeugte native Code ist (hoffentlich) effizient. Er wird in einem Speicher,

meist Cache genannt, aufgesammelt, so dass jede Funktion nur einmal übersetzt werden muss.

1.3 Allgemeine Literaturhinweise

Eine Übersicht über die Geschichte der Programmiersprachen und verschiedene Programmiersprachenkonzepte bieten u.a. [Seb05, Sco05, TN06]. Den Einsatz der Hardwarebeschreibungssprache VHDL beschreibt [Ped04]. Die Seitenbeschreibungssprache Postscript findet man in [Inc99]. Eine Übersicht über ActionScript zur Realisierung von Flash-Animationen enthält etwa [Hau06]. Die XML-Transformationsprache XSLT-2.0 wird ausführlich in [Kay04] erläutert. Für die XML-Anfragesprache XQuery kann man [KCD⁺03, Bru04] konsultieren. Einen Einstieg in gängige W3C-Standards rund um Web Services und Business Processes findet man etwa über [ACKM03]. Unterschiedliche Anwendungen für virtuelle Maschinen insbesondere im Bereich von Betriebssystemen behandelt [SN05].

Die Übersetzung imperativer Programmiersprachen

Wir beginnen unsere Darstellung mit der Übersetzung einer imperativen Programmiersprache. Als Beispiel wählen wir dafür einen Ausschnitt aus der Programmiersprache C. Als Zielsprache der Übersetzung wählen wir die Maschinensprache einer geeigneten virtuellen Maschine, die wir für diesen Zweck zusammen mit der Übersetzung entwerfen. Diese virtuelle Maschine nennen wir *C-Maschine* oder kurz: *CMa*.

2.1 Sprachkonzepte und ihre Übersetzung

Imperative Programmiersprachen besitzen u.a. die folgenden Konstrukte und Konzepte, die auf Konstrukte, Konzepte und Befehlsfolgen virtueller oder realer Rechner abgebildet werden müssen:

Variablen. Das sind Behälter für Datenobjekte, die ihren Inhalt (Wert) im Laufe der Programmausführung ändern können. Die Wertänderungen erfolgen durch die Ausführung von *Anweisungen* wie etwa Wertzuweisungen (assignments). Mehrere Variablen können in Aggregaten, Feldern (arrays) und Verbunden (records bzw. structs) zusammengefasst werden. Die aktuellen Werte der Variablen zu irgendeinem Zeitpunkt machen einen Teil des *Zustands* des Programms zu diesem Zeitpunkt aus. Variablen werden in Programmen mit *Namen* bezeichnet. Da auch Konstanten, Funktionen usw. mit Namen bezeichnet werden, sprechen wir von Variablenbezeichnungen, Konstantenbezeichnungen usw., wenn wir diese speziellen Arten von Namen unterscheiden wollen. Variablenbezeichnungen müssen Speicherzellen von Maschinen zugeordnet werden, die ihre jeweils aktuellen Werte enthalten. Enthält die Programmiersprache rekursive Funktionen mit lokalen Namen, so entstehen durch den Aufruf von Funktionen neue *Inkarnationen* der lokalen Variablenbezeichnungen; ihnen müssen dann jeweils neue Speicherzellen zugeordnet werden. Bei Verlassen der Prozedur werden die Zellen für diese Inkarnationen wieder freigegeben. Deshalb werden solche Sprachen mit Hilfe einer kellerartigen Speicherverwaltung implementiert.

Ausdrücke. Das sind aus Konstanten, Namen und Operatoren zusammengesetzte Terme, die, wenn man ihren Wert braucht, *ausgewertet* werden. Ihr Wert ist i.A. zustandsabhängig, da bei jeder Auswertung die aktuellen Werte der im Ausdruck enthaltenen Variablen zur Auswertung benutzt werden.

Explizite Angabe des Kontrollflusses. Der in den meisten imperativen Programmiersprachen existierende Sprungbefehl, **goto**, kann direkt in den unbedingten Sprungbefehl der Zielmaschine übersetzt werden. Höhere Kontrollkonstrukte wie bedingte (*if*) oder iterative (*while*, *do-while*, *for*) Anweisungen werden mit Hilfe bedingter Sprünge übersetzt. Ein bedingter Sprung folgt auf eine Befehlsfolge zur Auswertung einer Bedingung. Fallunterscheidungen (*case* bzw. *switch*) lassen sich auf manchen Zielmaschinen durch *indizierte* Sprünge effizient realisieren. Dabei wird die im Befehl angegebene Sprungadresse mit einem vorher berechneten Wert modifiziert.

Funktionen. Funktionen bzw. Prozeduren dienen der *funktionalen Abstraktion*, die aus einer evtl. komplexen Anweisung oder Anweisungsfolge eine neue Anweisung machen. Ein *Aufruf* dieser neu definierten Anweisung an einer Programmstelle aktiviert die in der Definition angegebene Folge von Anweisungen. Nach ihrer Abarbeitung wird an die Aufrufstelle zurückgekehrt. Hat die Funktion *formale Parameter*, so können durch die Angabe von *aktuellen Parametern* verschiedene *Inkarnationen* der Funktion aufgerufen werden. Zur Implementierung von Funktionen muss der Befehlssatz der Maschine einen Sprungbefehl bereitstellen, der seine Herkunft nicht vergisst, damit die Kontrolle an die Aufrufstelle zurückkehren kann. Der Rumpf der Funktion muss bei jeder Aktivierung (Aufruf) mit den aktuellen Parametern versorgt werden. Dies, zusammen mit der Erzeugung von Inkarnationen lokaler Namen, erfordert die oben erwähnte kellerartige Speicherorganisation, die häufig durch spezielle Maschineninstruktionen unterstützt wird.

2.2 Die Architektur der C-Maschine

Jede virtuelle Maschine stellt einen Satz *Instruktionen* zur Verfügung, die auf einer virtuellen Hardware ausgeführt werden. Diese virtuelle Hardware wird meist in Software *emuliert*. Der Ausführungszustand wird dabei in einer Menge von Datenstrukturen gespeichert, auf die die Instruktionen zugreifen und die vom *Laufzeitsystem* verwaltet werden.

Der Übersichtlichkeit halber werden wir die Architektur und die Instruktionen der C-Maschine schrittweise einführen, so wie wir sie für die Übersetzung der jeweils behandelten Konzepte in der Quellsprache brauchen. Zunächst wollen wir nur die benötigten Speichersegmente, einige Register und den *Hauptausführungszyklus* (Main Cycle) der C-Maschine einführen.

Die C-Maschine hat einen *Datenspeicher S* der Länge $\text{maxS} + 1$. An seinem unteren Ende, d.h. ab Adresse 0, liegt ein pulsierender *Keller* (engl. Stack). Das Register *SP* (Stack Pointer oder Kellerzeiger) zeigt jeweils auf die oberste belegte Zelle. Für alle Instruktionen der virtuellen Maschine vereinbaren wir die Konvention,

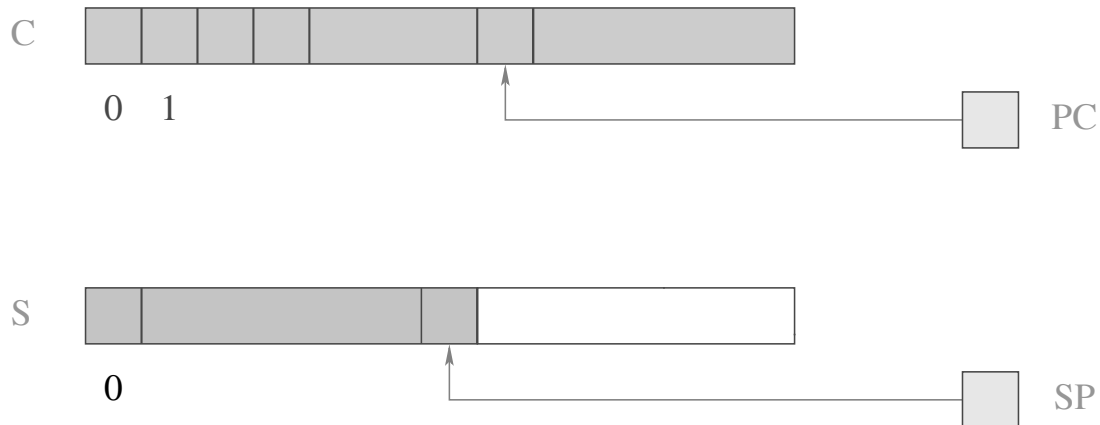


Abb. 2.1. Die Speicher der C-Maschine und die Register *PC* und *SP*

dass sie ihre Operanden oben auf dem Keller erwarten und ihre Ergebnisse auch dort hinterlassen. Zur Vereinfachung nehmen wir an, dass Werte der skalaren Typen in eine Speicherzelle des Datenspeichers passen. Als skalare Typen betrachten wir hier nur **int** und Zeigertypen, d.h. Adressen.

Der *Programmspeicher* der C-Maschine enthält das auszuführende Programm. Er hat die Länge $\max C + 1$. Jeweils ein Befehl der C-Maschine wird zur Ausführung in einer Zelle des Programmspeichers abgelegt. Der *Befehlszähler*, das Register *PC*, enthält jeweils die Adresse des nächsten auszuführenden Befehls. Dieser wird in ein *Instruktionsregister IR* geladen und anschließend ausgeführt. Vor der Ausführung wird der Inhalt des Befehlszählers *PC* um 1 erhöht, was bei sequentieller Ausführung *PC* auf den nächsten auszuführenden Befehl zeigen lässt. Ist der aktuelle Befehl ein Sprung, so überschreibt er den Inhalt des Befehlszählers *PC* mit seiner Zieladresse.

Also sieht der Hauptzyklus der C-Maschine folgendermaßen aus:

```
while (true) {
    IR  $\leftarrow$  C[PC]; PC++;
    execute (IR);
}
```

Am Anfang der Programmausführung enthält das Register *PC* den Wert 0. Die Programmausführung startet deshalb mit der Ausführung des Befehls $C[0]$. Die C-Maschine hält an, indem sie den Befehl **halt** ausführt. Die Ausführung dieses Befehls beendet den Ausführungszyklus und gibt die Kontrolle an das Betriebssystem zurück. Als Rückgabewert des Programms meldet sie dabei den Inhalt einer speziellen Speicherzelle zurück. Weil der Zugriff auf die Speicherzelle $S[0]$ verboten ist, nehmen wir hier an, dies sei die Zelle mit Adresse 1.

2.3 Einfache Ausdrücke und Wertzuweisungen

In diesem Abschnitt führen wir in die Übersetzung von arithmetischen und logischen Ausdrücken ein. Der Übersetzer muss für jeden Ausdruck eine Befehlssequenz erzeugen, deren Ausführung den (aktuellen) Wert des Ausdrucks ergibt. Betrachten wir als einfaches Beispiel die Übersetzung des Ausdrucks $(1 + 7) \cdot (2 + 5)$. Wie sieht eine Befehlsfolge aus, die diesen Ausdruck auswertet und sein Ergebnis oben auf dem Keller hinterlässt?

Besteht der Ausdruck nur aus einer Konstanten, z.B. 7, ist die Aufgabe leicht. Wir benötigen nur eine Instruktion, die einen gegebenen Wert oben auf den Keller schreibt. Zu diesem Zweck führen wir die Instruktion **loadc** q für beliebige Konstanten q ein (Abb. 2.2).

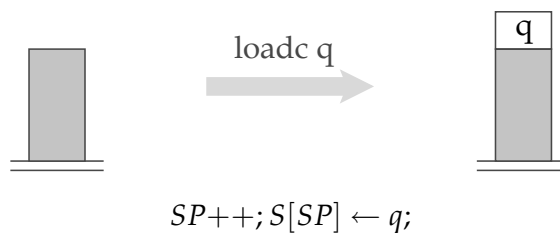


Abb. 2.2. Die Instruktion **loadc** q .

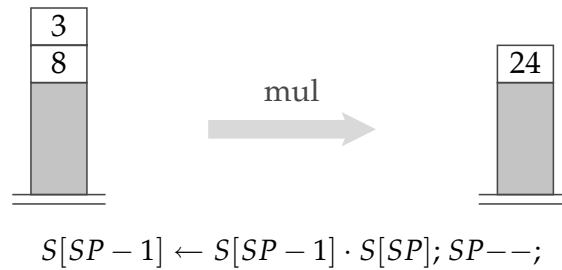
Die Instruktion **loadc** q benötigt keine weiteren Argumente. Sie legt als Wert die Konstante q oben auf dem Keller ab.

Für zusammengesetzte Ausdrücke besteht die Idee darin, zuerst die Werte der Unterausdrücke, hier $(1 + 7)$ und $(2 + 5)$, zu berechnen und ihre Werte jeweils in den oberen Kellerzellen zu hinterlassen, um dann den äußeren Operator, hier „ \cdot “, auf diese Werte anzuwenden. Die Anwendung des Operators sollte die Zwischenergebnisse auf dem Keller konsumieren und das Ergebnis des gesamten Ausdrucks wiederum oben auf dem Keller hinterlassen.

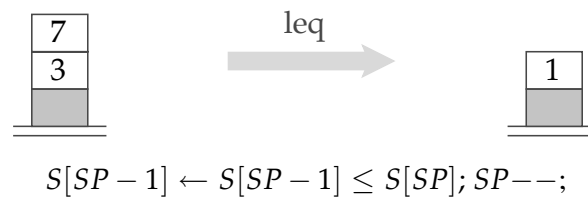
Ein solches rekursive Vorgehen gemäß der Struktur eines Programmfragments (hier: von Ausdrücken) werden wir immer wieder vorfinden. Dieses Vorgehen wird von unserer Entwurfsentscheidung unterstützt, dass insbesondere die arithmetischen, logischen und Vergleichsbefehle ihre Operanden jeweils oben auf dem Keller erwarten und durch die Ergebnisse des Befehls ersetzen. Wir demonstrieren dies in Abb. 2.3 anhand der **mul**-Instruktion, welche zwei ganzzahlige Operanden miteinander multipliziert.

Die Instruktion **mul** erwartet zwei Argumente oben auf dem Keller, multipliziert sie miteinander, konsumiert sie dabei und legt das Ergebnis oben auf dem Keller ab. Analog arbeiten auch die übrigen binären arithmetischen und logischen Instruktionen **add**, **sub**, **div**, **mod**, **and**, **or** wie auch die Vergleiche **eq**, **neq**, **le**, **leq**, **gr** und **geq**.

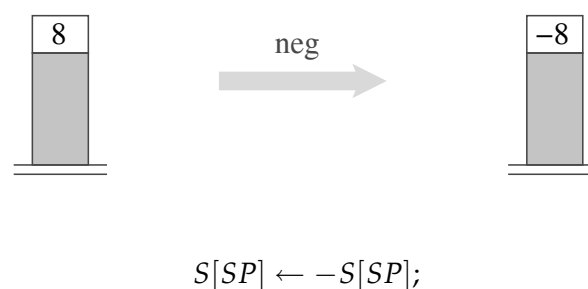
Vergleichsoperatoren erwarten ebenfalls zwei Operanden oben auf dem Keller, vergleichen sie und hinterlassen das Ergebnis des Vergleichs in der neuen obersten

Abb. 2.3. Die Instruktion **mul**.

Kellerzelle. Das Ergebnis sollte ein logischer Wert sein, also *true* oder *false* repräsentieren. Im Falle von C werden diese jedoch als ganzzahlige Werte dargestellt: 0 steht für *false*, alle anderen Werte für *true*. Abb. 2.4 zeigt, wie der **leq**-Befehl zwei ganze Zahlen vergleicht.

Abb. 2.4. Die Instruktion **leq**.

Unäre Befehle wie **neg** und **not** konsumieren dagegen nur einen Operanden. Da sie auch einen Wert als Ergebnis liefern, ersetzen sie somit den Wert in der obersten Kellerzelle. Als Beispiel zeigen wir den **neg**-Befehl, welcher das Vorzeichen einer Zahl umdreht (Abb. 2.5).

Abb. 2.5. Der Negationsbefehl **neg**.

Wir erzeugen für den Ausdruck $1 + 7$ die Befehlsfolge:

loadc 1; loadc 7; add

Abb. 2.6 zeigt, wie diese Befehlsfolge zur Laufzeit abgearbeitet wird.

Abb. 2.6. Abarbeitung der Befehlsfolge für $1 + 7$.

Welche Befehlsfolgen für eine Anweisung oder einen Ausdruck erzeugt werden, wird durch die Angabe von **code**-Funktionen spezifiziert. Diese Funktionen bekommen als Argument ein entsprechendes Programmfragment. Sie zerlegen dieses Argument rekursiv und setzen die für die Komponenten jeweils erzeugten Befehlsfolgen zur gesamten Befehlsfolge zusammen. Wir kümmern uns hier nicht um das Problem, wie ein C-Programm syntaktisch analysiert wird, d.h. wie seine syntaktische Struktur entdeckt wird. Ebenfalls verlassen wir uns darauf, dass die Typkorrektheit des Eingabeprogramms bereits geprüft wurde.

Den Variablen des Programms werden Zellen im Datenspeicher S zugeordnet, in denen ihre Werte abgepeichert werden. Die Adressen dieser Zellen benutzt der erzeugte Code, um die aktuellen Werte der Variablen zu laden oder neue Werte dort abzuspeichern (siehe Abb. 2.7).

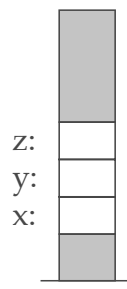


Abb. 2.7. Implementierung von Variablen.

Übersetzungsfunktionen benötigen darum als weiteres Argument eine Funktion ρ , die für jede Variable x die Adresse von x liefert. Die Funktion ρ heißt *Speicherbelegungsfunktion* oder *Adressumgebung* (Address Environment). Später werden wir sehen, dass die Adresse einer Variablen in Wirklichkeit eine *Relativadresse* ist, d.h. eine konstante Differenz zwischen zwei absoluten Adressen in S , nämlich der tatsächlichen Adresse der Zelle für diese Variable und der Anfangsadresse eines ganzen Speicherbereichs für alle Variablen, Parameter usw. der Funktion, in der die Variable deklariert ist. Für den Augenblick können wir $\rho(x)$ als die Adresse von x relativ zum Anfang von S auffassen.

In imperativen Sprachen können Variablen auf zwei Weisen verwendet werden. Betrachten wir z.B. die Zuweisung $x \leftarrow y + 1$. Von der Variablen y benötigen wir offenbar ihren Inhalt, um den Wert des Ausdrucks $y + 1$ ermitteln zu können. Von der

Variablen x dagegen interessiert der Inhalt nicht. Hier benötigen wir stattdessen die Adresse der zugehörigen Speicherzelle, um dort den neu berechneten Wert ablegen zu können. Wir schließen daraus, dass bei der Übersetzung von Wertzuweisungen eine Variablenbezeichnung, die auf der linken Seite einer Wertzuweisung steht, anders übersetzt werden muss als eine Variablenbezeichnung, die auf der rechten Seite steht. Von der Variablenbezeichnung links benötigt man die Adresse der ihr zugeordneten Zelle, ihren sogenannten *Linkswert* (L-Wert), um deren Inhalt zu überschreiben. Von der Variablenbezeichnung rechts braucht man ihren Wert, genauer: den *Rechtswert* (R-Wert). Wir indizieren deshalb die `code`-Funktionen mit L bzw. R; entsprechend erzeugt `codeL` Befehle zur Berechnung des L-Wertes und `codeR` Befehle zur Berechnung des R-Wertes. Die Funktion `code` (ohne Index) übersetzt Anweisungen. Beachten Sie jedoch schon jetzt, dass zwar jeder Ausdruck über einen R-Wert, nicht aber jeder Ausdruck auch über einen L-Wert verfügt. Ein einfaches Beispiel ist etwa der Ausdruck $y + 1$. Den Wert dieses Ausdrucks stellen wir uns als nur temporär auf dem Keller vorhanden und deshalb nicht adressierbar vor.

Die Abbildung 2.8 enthält einen Ausschnitt aus der Definition der Übersetzungsfunktionen `codeR` und `codeL` für einige Ausdrücke. Den L-Wert einer Variablen x

```

codeR (e1 + e2) ρ = codeR e1 ρ
                      codeR e2 ρ
                      add
                      // analog für die anderen binären Operatoren

codeR (−e) ρ      = codeR e ρ
                      neg
                      // analog für andere unäre Operatoren

codeR q ρ         = loadc q

codeL x ρ         = loadc ρ(x)

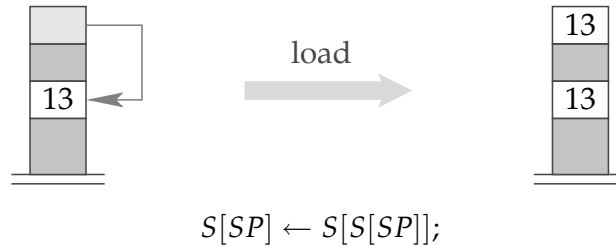
codeR x ρ         = codeL x ρ
                      load

```

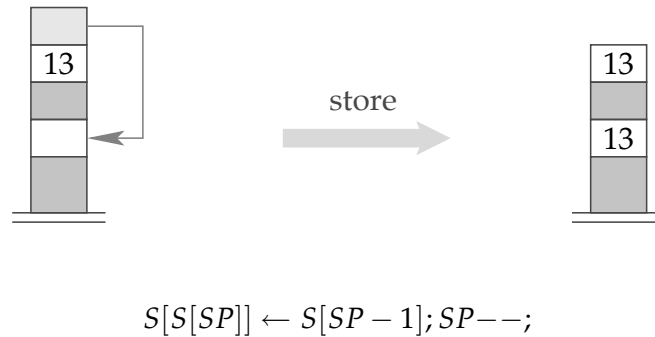
Abb. 2.8. Ein Ausschnitt der Definition von `codeR` und `codeL`.

können wir direkt aus der Adressumgebung ablesen. Um den R-Wert von x zu ermitteln, benötigen wir jedoch einen neuen Befehl, der uns erlaubt, den Inhalt einer Speicherzelle, deren Adresse oben auf dem Keller liegt, auf den Keller zu laden. Dazu dient die neue Instruktion **load** (Abb. 2.9).

In der Programmiersprache C sind Zuweisungen wie $x \leftarrow y + 1$ ebenfalls *Ausdrücke*. Der Wert dieses Ausdrucks ist dabei der Wert der rechten Seite der Zuweisung. Der R-Wert der linken Seite der Zuweisung ändert sich dabei mithilfe eines *Seiteneffekts* der Auswertung des Ausdrucks. Zur Implementierung der Zuweisung benötigen wir eine Instruktion **store** (Abb. 2.10). Die Instruktion **store** erwartet zwei

Abb. 2.9. Die Instruktion **load**.

Argumente auf dem Keller: einen Wert w und darüber eine Adresse a . Den Wert w schreibt sie an der Adresse a in den Speicher und liefert ihn gleichzeitig als Wert auf dem Keller zurück. In einer Adressumgebung $\rho = \{x \mapsto 4, y \mapsto 7\}$ berechnet die

Abb. 2.10. Die Instruktion **store**.

folgende Folge von Instruktionen den R-Wert von $x \leftarrow y + 1$:

loadc 7; load; loadc 1; add; loadc 4; store

Zuerst wird der Wert der rechten Seite ermittelt. Dann folgt eine Instruktionsfolge zur Berechnung des L-Werts der linken Seite, in unserem Fall **loadc 4**. Die Zuweisung selbst wird schließlich von der Instruktion **store** ausgeführt. Allgemein können wir also eine Zuweisung wie folgt übersetzen:

$$\begin{aligned} \text{code}_R(x \leftarrow e) \rho &= \text{code}_R e \rho \\ &\quad \text{code}_L x \rho \\ &\quad \text{store} \end{aligned}$$

Beispiel 2.3.1 Sei ein Programm mit den drei **int**-Variablen a, b, c gegeben. Die Speicherbelegungsfunktion ρ bilde a, b, c auf die Adressen 5, 6 bzw. 7 ab. Die Übersetzung der Wertzuweisung $a \leftarrow (b + (b \cdot c))$ geschieht folgendermaßen:

$$\begin{aligned}
& \text{code } (a \leftarrow (b + (b \cdot c))) \rho \\
&= \text{code}_R (b + (b \cdot c)) \rho; \text{code}_L a \rho; \text{store} \\
&= \text{code}_R b \rho; \text{code}_R (b \cdot c) \rho; \text{add}; \text{code}_L a \rho; \text{store} \\
&= \text{loadc } 6; \text{load}; \text{code}_R (b \cdot c) \rho; \text{add}; \text{code}_L a \rho; \text{store} \\
&= \text{loadc } 6; \text{load}; \text{code}_R b \rho; \text{code}_R c \rho; \text{mul}; \text{add}; \text{code}_L a \rho; \text{store} \\
&= \text{loadc } 6; \text{load}; \text{loadc } 6; \text{load}; \text{code}_R c \rho; \text{mul}; \text{add}; \text{code}_L a \rho; \text{store} \\
&= \text{loadc } 6; \text{load}; \text{loadc } 6; \text{load}; \text{loadc } 7; \text{load}; \text{mul}; \text{add}; \text{loadc } 5; \text{store}
\end{aligned}$$

□

An unseren Beispielen fällt auf, dass bestimmte Routineaufgaben sich wiederholen und damit zu immer ähnlichen Befehlsfolgen führen. So muss die Übersetzung oft Instruktionsfolgen erzeugen, die von einer konstanten (d.h. zur Übersetzungszeit bekannten) Adresse einen Wert laden bzw. an diese Adresse einen Wert schreiben. Als Optimierung könnten wir deshalb für diese Aufgaben Spezialbefehle einführen:

$$\begin{aligned}
\text{loada } q &= \text{loadc } q \\
&\quad \text{load} \\
\text{storea } q &= \text{loadc } q \\
&\quad \text{store}
\end{aligned}$$

Beachten Sie, dass solche Spezialbefehle tatsächlich zu einer Effizienzverbesserung führen: einerseits wird natürlich der erzeugte Code kürzer; andererseits kann eine Implementierung etwa des Befehls **loada** 7 effizienter vorgehen, als erst die Konstante 7 zu erzeugen, auf den Keller zu laden, um dann die 7 durch den Wert der Speicherzelle mit Adresse 7 zu überschreiben.

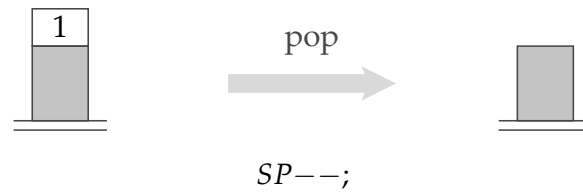
2.4 Anweisungen und Anweisungsfolgen

In C gilt: ist e ein Ausdruck, dann ist e ; eine *Anweisung* (Statement). Anweisungen liefern keinen Wert zurück. Folglich muss der Wert des Kellerzeigers SP vor und nach der Ausführung der erzeugten Befehlsfolge gleich sein. Deshalb übersetzen wir:

$$\begin{aligned}
\text{code } (e;) \rho &= \text{code}_R e \rho \\
&\quad \text{pop}
\end{aligned}$$

Dabei ist **pop** eine Instruktion, die das oberste Element vom Keller entfernt (Abb. 2.11).

Sind wir erst einmal in der Lage, Code für eine einzelne Anweisung zu generieren, ist es leicht, auch Code für Folgen von Anweisungen zu generieren. Wir konkatenieren die Codesequenzen für die einzelnen Anweisungen in der Folge:

Abb. 2.11. Die Instruktion **pop**.

```

code (s ss) ρ = code s ρ
               code ss ρ
               // s eine Anweisung, ss eine Folge von Anweisungen
code ε ρ      =      // eine leere Folge von Befehlen

```

2.5 Bedingte und iterative Anweisungen

Als nächstes wagen wir uns an die Übersetzung von bedingten und iterativen Anweisungen (Schleifen). Wir geben Übersetzungsschemata an für einseitige und zweiseitige *if*-Anweisungen:

```

if (e) s
if (e) s1 else s2

```

und für *while*- und *for*-Schleifen:

```

while (e) s
for (e1; e2; e3) s

```

wobei e, e_i jeweils Ausdrücke und s, s_i jeweils einzelne Anweisungen bzw. zu einem Block zusammengefasste Anweisungsfolgen sind.

Um von der linearen Ausführungsreihenfolge abzuweichen, benötigen wir geeignete Sprungbefehle. *Unbedingte* Sprünge setzen stets die Programmausführung an einer anderen Stelle fort. *Bedingte* Sprünge führen nur zu einer Abweichung von der sequentiellen Programmausführung, wenn eine bestimmte Bedingung zutrifft. In unserem Fall besteht diese Bedingung in einem Vergleich des obersten Elements im Keller mit 0 (Abb. 2.12). Statt absoluter Codeadressen könnte man alternativ auch *relative* Adressen als Sprungziele benutzen, d.h. in den Sprungbefehlen die Sprungziele stets relativ zum aktuellen *PC* angeben. Dies hätte den Vorteil, dass typischerweise kleinere Adressen ausreichen. Auch wird der Code leichter *relokierbar*, d.h. er kann an beliebiger Stelle im Speicher stehen.

In den Übersetzungsschemata für bedingte Anweisungen und Schleifen benötigen wir ein neues Hilfsmittel. Wir markieren Befehle oder auch das Ende des Schemas durch symbolische Marken, die wir in Sprungbefehlen als Sprungziele verwenden. Ein Vorkommen einer solchen Marke steht für die Adresse, welche der Befehl erhält bzw. schon erhalten hat, der diese Marke trägt. In einem zweiten Durchlauf

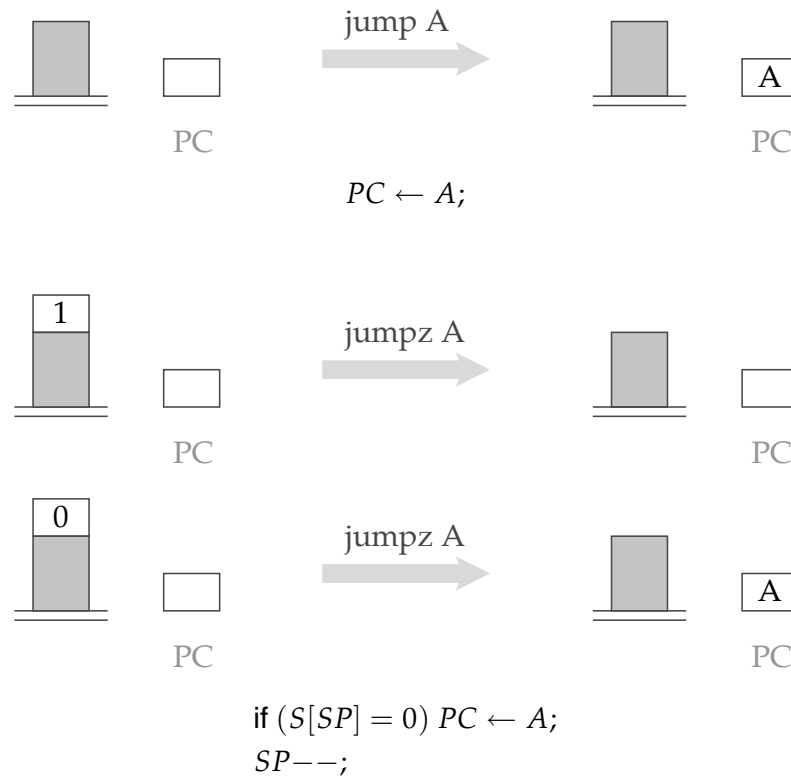


Abb. 2.12. Die Sprungbefehle **jump** und **jumpz A**

folgend auf die Codeerzeugung können die Marken dann durch absolute Codeadressen ersetzt werden.

Betrachten wir zuerst eine einseitige bedingte Anweisung s der Form **if** (e) s' . Bei der Codeerzeugung gehen wir so vor, dass wir zuerst den Code zur Auswertung von e und s' hintereinander in den Programmspeicher legen und dabei Sprungbefehle so einfügen, dass ein korrekter Kontrollfluss gewährleistet ist (Abb. 2.13). In diesem Fall heißt das, dass wir nach der Bedingung einen bedingten Sprung *hinter* die gesamte Anweisung einfügen müssen: wird die Bedingung e zur Laufzeit zu 0

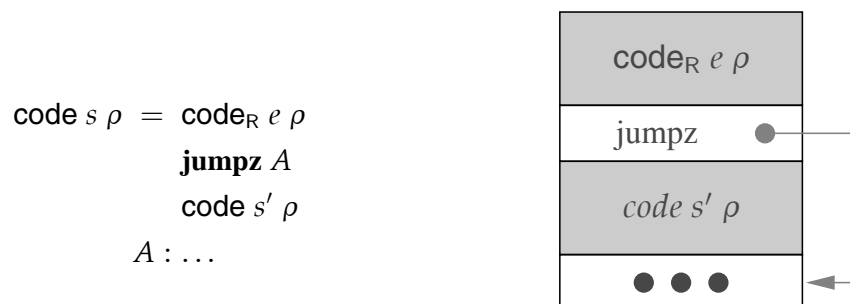


Abb. 2.13. Codeerzeugung für einseitige bedingte Anweisungen.

ausgewertet, wird die Anweisung sofort verlassen. Andernfalls wird die Instrukti-
onsfolge für die Anweisung s' ausgeführt.

Die gleiche Strategie wenden wir auch zur Codeerzeugung für eine zweiseitige
bedingte Anweisung s der Form **if** (e) s_1 **else** s_2 an: erst legen wir die Codesequen-
zen für e , s_1 und s_2 in den Programmspeicher. Dann fügen wir geeignete Sprünge
ein, um einen korrekten Kontrollfluss zu garantieren (Abb. 2.14). Nach der Bedin-

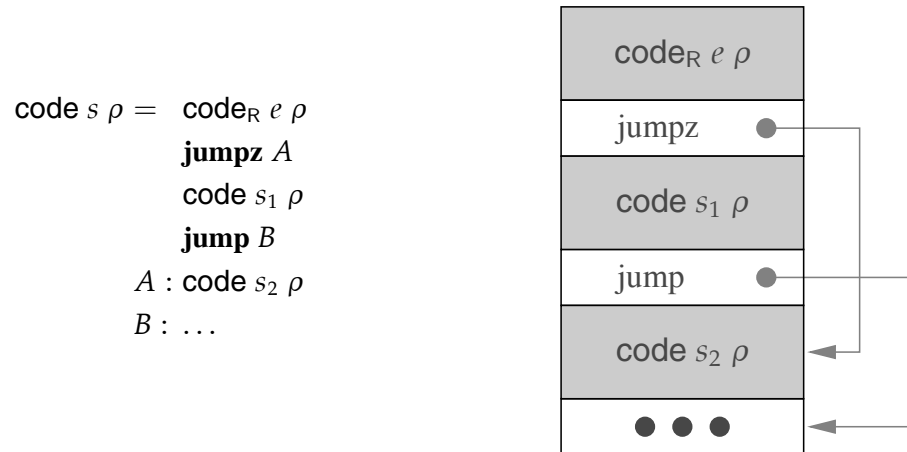


Abb. 2.14. Codeerzeugung für zweiseitige bedingte Anweisungen.

gung fügen wir einen bedingten Sprung an den Anfang des *else*-Teils s_2 ein. Damit
bei Auswertung der Bedingung e zu einem Wert ungleich 0 nicht nach Ausführung
des Codes für s_1 auch noch der Code für s_2 ausgeführt wird, müssen wir hinter den
Code für s_1 einen unbedingten Sprung hinter die gesamte Anweisung einfügen.

Beispiel 2.5.1 Sei $\rho = \{x \mapsto 4, y \mapsto 7\}$ und die bedingte Anweisung s von
der Form:

$$\begin{aligned} & \text{if } (x > y) \\ & \quad x \leftarrow x - y; \\ & \text{else } y \leftarrow y - x; \end{aligned}$$

Dann liefert $\text{code } s \ \rho$:

loada 4	loada 4	A: loada 7
loada 7	loada 7	loada 4
gr	sub	sub
jumpz A	storea 4	storea 7
	pop	pop
	jump B	B: ...

□

Betrachten wir eine *while*-Schleife s der Form **while** (e) s' . Die für s generierte Instruktionsfolge zeigt Abb. 2.15. Hinter den Code für die Bedingung fügen wir

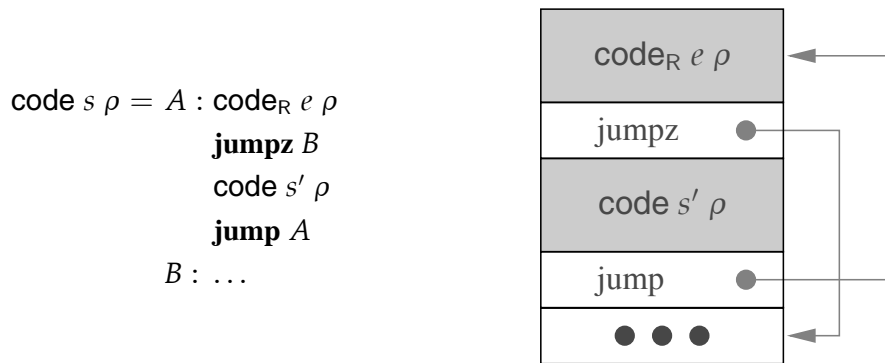


Abb. 2.15. Die Übersetzung einer *while*-Schleife.

einen bedingten Sprung aus der Schleife heraus ein, während am Ende des Codes für den Rumpf ein unbedingter Sprung zurück an den Anfang des Bedingungscode der Schleife eingefügt wird.

Beispiel 2.5.2 Sei $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ eine Adressumgebung und s die Anweisung:

while $(a > 0)$ $\{c \leftarrow c + 1; a \leftarrow a - b;\}$

Dann liefert $\text{code } s \rho$ die Folge:

<p>A: loada 7</p> <p> loadc 0</p> <p> gr</p> <p> jumpz B</p>	<p> loada 9</p> <p> loadc 1</p> <p> add</p> <p> storea 9</p> <p> pop</p>	<p> loada 7</p> <p> loada 8</p> <p> sub</p> <p> storea 7</p> <p> pop</p> <p> jump A</p>	<p>B: ...</p>
--	--	---	------------------

□

Eine Komplikation bei der Codeerzeugung tritt auf, wenn der Rumpf der Schleife *break*- oder *continue*-Anweisungen enthält. Ein **break** muss aufgefasst werden als ein unbedingter Sprung aus der Schleife heraus. Ein **continue** dagegen springt nur an das Ende des Rumpfs. Für eine korrekte Codeerzeugung muss deshalb die Übersetzungsfunktion **code** zusätzlich die aktuellen Sprungziele für **break** oder **continue** mitverwalten. Für Details verweisen wir auf Aufgabe 6.

Eine *for*-Schleife s der Form **for** $(e_1; e_2; e_3)$ s' ist äquivalent zu der Anweisungsfolge:

$e_1; \text{ while } (e_2) \{s' e_3;\}$

– sofern s' keine *continue*-Anweisung enthält. In diesem Fall übersetzen wir:

```

code  $s \rho$  = codeR  $e_1$ 
      pop
       $A : \text{code}_R e_2 \rho$ 
      jumpz  $B$ 
      code  $s' \rho$ 
      codeR  $e_3 \rho$ 
      pop
      jump  $A$ 
       $B : \dots$ 

```

In Anwesenheit von *continue*-Anweisungen im Rumpf s' der Schleife können wir bei der Übersetzung im Prinzip genauso vorgehen. Wir müssen nur darauf achten, dass wir jedes in s' vorkommende **continue** als unbedingten Sprung an das Ende von s' interpretieren, d.h. vor die Instruktionsfolge für e_3 .

Betrachten wir nun die *switch*-Anweisung der Programmiersprache C. Die Idee dieser Anweisung ist, eine Mehrfachverzweigung abhängig vom Wert eines Selektorausdrucks effizient zu unterstützen. Zur Vereinfachung nehmen wir an, dass in der Anweisung nur zwischen den Fällen 0 bis $k - 1$ für eine Konstante k ausgewählt wird. Alle anderen Werte des Selektionsausdrucks sollen die *default*-Alternative auswählen. Zur weiteren Vereinfachung nehmen wir an, dass die Fälle in aufsteigender Reihenfolge angeordnet sind und jeder Fall mit einem **break** abgeschlossen wird. Unsere *switch*-Anweisung s hat also das Aussehen:

```

switch ( $e$ ) {
  case 0:   $ss_0$  break;
  case 1:   $ss_1$  break;
       $\vdots$ 
  case  $k - 1$ :   $ss_{k-1}$  break;
  default:   $ss_k$ 
}

```

wobei die ss_i Folgen von Anweisungen sind. Eine Möglichkeit, *switch*-Anweisungen zu übersetzen, benutzt *indizierte Sprünge*. Ein indizierter Sprung ist ein Sprung, bei dem die angegebene Zieladresse um einen unmittelbar vorher berechneten Wert erhöht wird. Wir benutzen dazu den Befehl **jumpi** (Abb. 2.16). Dann ergibt sich für die *switch*-Anweisung s die Instruktionsfolge:

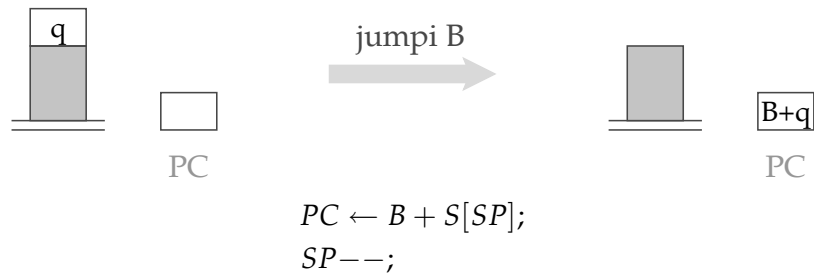


Abb. 2.16. Der indizierte Sprung **jumpi**.

$\text{code } s \rho = \text{code}_R e \rho$ $\text{check } 0 k B$	$C_0:$ $\text{code } ss_0 \rho$ $\text{jump } D$ \dots $C_k:$ $\text{code } ss_k \rho$ $\text{jump } D$	$B:$ $\text{jump } C_0$ \dots $\text{jump } C_k$ $D:$ \dots
---	---	--

Das *Makro* $\text{check } 0 k B$ überprüft, ob der R-Wert von e im Intervall $[0, k]$ liegt, und führt einen indizierten Sprung in die Tabelle durch, die ab Adresse B im Programmspeicher C angelegt ist. Die Sprungtabelle enthält direkte Sprünge zu den jeweiligen Alternativen. Am Ende jeder Alternative steht ein Sprung hinter die *switch*-Anweisung. Eine Implementierung des Makros check könnte etwa so aussehen:

$\text{check } 0 k B =$ dup $\text{loadc } 0$ geq $\text{jumpz } A$	dup $\text{loadc } k$ leq $\text{jumpz } A$	$A:$ pop $\text{loadc } k$ $\text{jumpi } B$
---	--	---

Der ganzzahlige Wert oben auf dem Keller, den wir zur Fallunterscheidung benutzen, wird in zwei Vergleichen benötigt, um zu überprüfen, dass er tatsächlich innerhalb des Intervalls $[0, k - 1]$ liegt. Erst wenn wir dies überprüft haben, können wir ihn zur Indizierung der Sprungtabelle benutzen. Weil jedoch bei unserer Maschine jeder Vergleich den Wert konsumiert, müssen wir ihn vor jedem Vergleich zuerst *duplizieren*. Dazu dient der Befehl **dup** (Abb. 2.17). Die Idee zur Implementierung des Makro

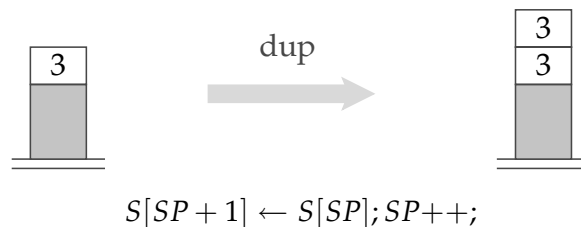


Abb. 2.17. Die Instruktion **dup**.

ist dann ganz einfach. Ist der R-Wert von e kleiner als 0 oder größer als k , ersetzen wir ihn vor dem indizierten Sprung durch k . Für Werte i im Intervall $[0, k - 1]$ führt der indizierte Sprung auf einen unbedingten Sprung an die Anfangsadresse der i -ten Alternative. Für den Wert k springt der indizierte Sprung dagegen den unbedingten Sprung auf die *default*-Alternative an.

In unserem Übersetzungsschema haben wir die Sprungtabelle ganz ans Ende der Instruktionsfolge für die *switch*-Anweisung gestellt. Alternativ hätten wir sie natürlich auch direkt hinter dem Makro *check* platzieren können. Dadurch hätten wir ein paar unbedingte Sprünge einsparen können. Eventuell hätte die Codeerzeugung die Anweisung aber zweimal durchlaufen müssen, um die benötigten Anfangsadressen der verschiedenen Alternativen zu sammeln.

Die Idee unserer Behandlung der sehr vereinfachten *switch*-Anweisung lässt sich verallgemeinern. Ist der kleinste vorkommende Wert etwa u (anstatt 0), vermindern wir den R-Wert von e um u , bevor wir ihn als Index benutzen. Auch ist eine strikt aufsteigende Anordnung der einzelnen Selektorwerte nicht wirklich erforderlich. Selbstverständlich dürfen auch Lücken im Intervall der möglichen Selektorwerte auftreten. Die entsprechenden Einträge der Sprungtabelle füllen wir dann mit unbedingten Sprüngen auf die *default*-Alternative. Probleme treten auf, wenn das Intervall der möglichen Selektorwerte sehr groß ist und gleichzeitig nur sehr wenig Werte daraus wirklich auftreten. In Aufg. 8 können Sie Lösungsansätze für diesen Spezialfall entwickeln.

2.6 Speicherbelegung für Variablen einfachen Typs

In diesem Abschnitt führen wir einige sehr wichtige Begriffe des Übersetzerbaus ein, nämlich die Begriffe *Übersetzungszeit* und *Laufzeit*, *statisch* und *dynamisch*. Zur *Übersetzungszeit* wird ein vorgelegtes C-Programm in ein CMa-Programm übersetzt. Zur *Laufzeit* wird dieses erzeugte C-Programm mit Eingabedaten ausgeführt. *Statisch* sind alle Informationen über ein C-Programm, die zur Übersetzungszeit allein aus diesem Programm ersichtlich sind oder aus ersichtlicher Information berechnet oder erzeugt werden können. *Dynamisch* sind all die Informationen, die erst zur Laufzeit durch das Ausführen des erzeugten CMa-Programms mit Eingabedaten verfügbar werden.

Wir haben schon einige Beispiele für statische und für dynamische Informationen über C-Programme kennengelernt. Statisch etwa sind die Zieladressen von bedingten oder unbedingten Sprüngen, denn sie werden schließlich mithilfe der *code*-Funktion aus dem Quellprogramm berechnet. Dies gilt natürlich für das ganze erzeugte CMa-Programm. Also ist dieses auch statisch. Dynamisch sind i.A. die Werte von Variablen, also auch die Werte von Ausdrücken, in denen Variablen auftreten. Diese Werte hängen i.A. von Eingabewerten des Programms ab, welche erst zur Laufzeit zur Verfügung stehen. Da also die Werte der Bedingungen in Anweisungen dynamisch sind, ist es auch dynamische Information, wie der Kontrollfluss nach der Auswertung der Bedingung aussieht.

Um die Zuordnung von im Quellprogramm deklarierten Variablen zu Speicherzellen vorzunehmen, müssen wir an unsere Annahme über die Größe der Speicherzellen und unsere (Speicher-) Spendierfreudigkeit erinnern. Jeder Variablen der einfachen Typen **int** (oder auch **float**, **char**, sowie vom Aufzählungs- oder Mengentyp, sofern wir sie hier behandeln) und Zeigervariablen ordnen wir eine Speicherzelle zur Aufnahme ihrer Werte zu. Wir versuchen also nicht, wie es in realen Übersetzern geschieht, „kleine Werte“ zu mehreren in ein Wort zu packen. Wir können also augenblicklich nur Programme übersetzen, in denen die Genauigkeit durch die Wortlänge der C-Maschine beschränkt ist. Diese Wortlänge ist dabei auch nicht festgelegt. Sie muss nur in der Lage sein, *int*-Werte und Adressen aufzunehmen. Dann ergibt sich ein recht einfaches Schema zur Speicherbelegung. Den Variablen im Deklarationsteil des Programms – wir betrachten im Augenblick noch Programme ohne Blöcke und Funktionen – ordnen wir in der Reihenfolge ihres Auftretens konsekutive Adressen am Anfang des Kellerspeichers zu. Die erste zugeteilte Adresse ist aus später ersichtlichen Gründen nicht 0 sondern 1. Die zugeteilten Adressen nennen wir aus Gründen, die ebenfalls klar werden, wenn wir Funktionen und Blöcke betrachten, *relative Adressen*. Die (eigentlich) absolute Adresse 1 fassen wir also auf als die Adresse 1 relativ zur Basis 0.

Betrachten wir den Variablendeklarationsteil eines C-Programms, wobei die vorkommenden Typen alle einfach seien. Er hat dann die Form:

$$t_1 x_1; \dots; t_k x_k;$$

Die oben geschilderte Speicherbelegungsstrategie würde dann die Funktion ρ , die die Zuordnung von Variablen zu Relativadressen festhält, folgendermaßen definieren:

$$\rho(x_i) = i \text{ für } 1 \leq i \leq k.$$

Man sieht leicht, dass die so zugeteilten relativen Adressen statische Größen sind; denn sie ergeben sich (auf sehr einfache Weise) aus dem Quellprogramm, nämlich aus den Positionen im Variablendeklarationsteil.

Diese Adressen liegen natürlich in dem Bereich des Speichers, den wir für den Keller der C-Maschine reserviert haben. Wenn Funktionen und Prozeduren behandelt werden, wird klar, dass wir eigentlich von zwei ineinander enthaltenen Kellern reden, nämlich einem „großen“, der aus den Datenbereichen aller jeweils aktiven Prozeduren besteht und somit wächst bzw. schrumpft, wenn eine Funktion betreten bzw. verlassen wird, und einem „kleinen“ zu jeder aktiven Funktion, der die während der Verarbeitung der Anweisungen des Rumpfes anfallenden Zwischenergebnisse aufnimmt. Nur mit diesem zweiten haben wir bisher Bekanntschaft gemacht, etwa bei der Verarbeitung von Wertzuweisungen. Die Zuteilung von Speicherzellen bzw. Adressen an deklarierte Variablen legt den Aufbau des Datenbereichs fest.

2.7 Speicherbelegung für Felder und Strukturen

Die Programmiersprache C stellt nur *statische Felder* zur Verfügung. Betrachten wir die folgende Deklaration eines Feldes *a*:


```
int a [11];
```

Wieviele Speicherzellen wird das Feld a belegen? Offensichtlich besteht a aus den 11 Komponenten:

$$a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10]$$

von denen jede nach unserer obigen Annahme eine Zelle belegt. Damit benötigt das Feld also elf Speicherzellen für seine Komponenten. Diese Elemente legen wir konsequent in den Keller, wobei wir die Anfangsadresse, d.h. diejenige der Komponente $a[0]$ in der Speicherbelegungsfunktion für a vermerken.

Im Allgemeinen müssen natürlich die Komponenten keineswegs stets von einem einfachen Typ sein, sondern können selbst wieder zusammengesetzt sein. Deshalb benötigen wir zur Konstruktion der Speicherbelegung für mehrere Variablen eine Hilfsfunktion, die uns für jeden Typ t die Anzahl der benötigten Speicherzellen berechnet:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ einfach} \\ k \cdot |t'| & \text{falls } t \equiv t' [k] \end{cases}$$

Diese Funktion wird von der Programmiersprache C als Hilfsfunktion `sizeof` zur Verfügung gestellt.

Auch in Anwesenheit von komplexeren Typen wollen wir die deklarierten Variablen konsequent im Speicher ablegen. Dann ergibt sich für eine Folge d von Deklarationen der Form $t_1 x_1; \dots; t_k x_k$; eine Speicherbelegungsfunktion ρ mit:

$$\begin{aligned} \rho(x_1) &= 1 \\ \rho(x_i) &= \rho(x_{i-1}) + |t_{i-1}| && \text{für } i > 1 \end{aligned}$$

Machen Sie sich an dieser Stelle klar, dass diese Speicherbelegungsfunktion tatsächlich zur Übersetzungszeit, d.h. aus dem gegebenen Deklarationsteil des C-Programms berechnet werden kann. Entsprechend könnten wir zum Abspeichern, sagen wir des Wertes 42, in die Komponente $a[0]$ des obigen Feldes a die folgende Befehlsfolge benutzen:

```
loadc 42; loadc  $\rho(a)$ ; store; pop
```

wobei die Adresse $\rho(a)$ statisch bekannt ist.

Interessant wird es, wenn wir die Wertzuweisung $a[i] \leftarrow 42$; übersetzen sollen, worin i eine *int*-Variable ist. Die Variable i erhält ihren Wert erst zur Ausführungszeit des übersetzten Programms. Darum müssen nun Befehle erzeugt werden, welche zuerst den aktuellen Wert von i ermitteln. Dann kann die richtige Komponente des Felds selektiert werden, indem der notwendige Betrag zu der Anfangsadresse $\rho(a)$ addiert wird:

```
loadc 42; loadc  $\rho(a)$ ; loadc  $\rho(i)$ ; load; add; store; pop
```

Allgemeiner sei a ein Ausdruck, der ein Feld von Komponenten des Typs t repräsentiert. Um die Anfangsadresse der Komponente $a[e]$ zu bestimmen, muss erst die

Anfangsadresse des Felds a ermittelt werden. Dann wird die Nummer der selektierten Komponente bestimmt, indem der R-Wert von e berechnet wird. Dieser Index muss mit dem Platzbedarf für jede einzelne Komponente skaliert werden, bevor er zur Anfangsadresse des Feldes addiert wird, um die gewünschte Anfangsadresse des Ausdrucks $a[e]$ zu erhalten. Insgesamt muss also der Wert des Ausdrucks

$$(L\text{-Wert von } a) + |t| * (R\text{-Wert von } e)$$

berechnet werden. Deshalb erweitern wir die Übersetzungsfunktion `codeL` auf indizierte Feldausdrücke durch:

$$\begin{aligned} \text{code}_L a[e] \rho &= && \text{code}_L a \rho \\ &&& \text{code}_R e \rho \\ &&& \text{loadc } |t| \\ &&& \text{mul} \\ &&& \text{add} \end{aligned}$$

Ist der L-Wert eines indizierten Feldausdrucks $a[e]$ bekannt, so kann leicht auch der zugehörige R-Wert bestimmt werden, indem der Inhalt der adressierten Speicherzelle geladen wird. Dies funktioniert jedoch nur, wenn die Komponenten des indizierten Felds genau in eine Speicherzelle passen. Wie zusammengesetzte R-Werte behandelt werden können, werden wir gleich betrachten.

Zuvor betrachten wir aber das Problem der Speicherbelegung und Adressierung von *Verbunden* oder Strukturen. Sei etwa die Verbundvariable x deklariert durch:

struct t { **int** a ; **int** b ; } x ;

Dann ordnen wir wie bisher der Variablen x die Adresse der ersten freien Speicherzelle zu. Für die Komponenten von x vergeben wir Adressen *relativ* zum Anfang der Struktur, hier $a \mapsto 0$, $b \mapsto 1$. Diese Relativadressen hängen nur von dem Typ t ab. Wir sammeln sie deshalb in der Funktion `offsets`, die Paaren (t, c) von Verbundtypen und ihren Komponenten die zugehörigen Relativadressen zuordnet.

Sei allgemein t ein Verbundtyp von der Form **struct** $\{t_1 \ c_1; \dots t_k \ c_k\}$. Dann definieren wir:

$$\begin{aligned} \text{offsets}(t, c_1) &= 0 \quad \text{und} \\ \text{offsets}(t, c_i) &= \text{offsets}(t, c_{i-1}) + |t_{i-1}| \quad \text{für } i > 1 \end{aligned}$$

Dabei ergibt sich die Größe des Verbundtyps t als Summe der Größen seiner Komponenten:

$$|t| = \sum_{i=1}^k |t_i|$$

Die Adressierung von Verbundkomponenten geschieht analog der Adressierung von Feldkomponenten in den folgenden Schritten:

1. Laden der Anfangsadresse des Verbundes;
2. Erhöhung der Adresse um die Relativadresse der Komponente.

Damit hat man die Adresse, also einen verwendbaren L-Wert. Allgemein sei e ein Ausdruck von dem Verbundtyp t , welcher eine Komponente c hat. Dann wird zur Berechnung des L-Werts von $e.c$ der folgende Code erzeugt:

$$\text{code}_L(e.c) \rho = \begin{array}{l} \text{code}_L e \rho \\ \quad \text{loadc } m \\ \quad \text{add} \end{array}$$

wobei $m = \text{offsets}(t, c)$. Wie kann der R-Wert einer Komponente ermittelt werden? Für Komponenten, deren Typ zusammengesetzt ist, reicht das Laden des Inhalts der adressierten Zelle offenbar nicht aus. Stattdessen muss ein ganzer Block oben auf den Keller geladen werden. Um mit dieser Situation zurecht zu kommen, verallgemeinern wir die Instruktion **load** zu Instruktionen **load** m für beliebige nicht-negative Werte m . Diese legen den Inhalt von m aufeinanderfolgenden Zellen ab der Adresse, die oben auf dem Keller liegt, oben auf den Keller (Abb. 2.18). Insbesondere ist

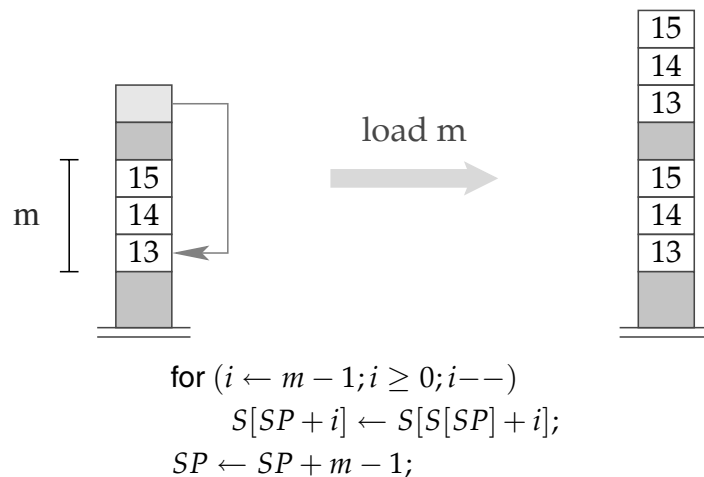


Abb. 2.18. Die Instruktion **load** m .

natürlich die Instruktion **load** 1 äquivalent zu unserer bisherigen Instruktion **load**. Damit erhalten wir als Übersetzungsschema zur Berechnung des R-Werts eines Ausdrucks e vom Verbundtyp der Größe m :

$$\text{code}_R(e) \rho = \begin{array}{l} \text{code}_L e \rho \\ \quad \text{load } m \end{array}$$

Mit Bedacht haben wir die Anwendbarkeit dieses Übersetzungsschemas auf Ausdrücke vom Verbundtyp beschränkt. Felder sind natürlich i.A. ebenfalls zusammengesetzt. Aus historischen Gründen ist der R-Wert eines Felds a in C jedoch *nicht* die

Folge der R-Werte der Komponenten von a , sondern die *Anfangsadresse* von a . Dies liegt daran, dass entsprechend der C-Philosophie, der R-Wert des Felds a aufgefasst wird als *Zeiger* auf den Speicherbereich, in dem die Komponenten von a liegen. Der R-Wert des Felds a ist deshalb vom Typ $t *$, falls t der Typ der Komponenten von a ist. Ist also e ein Ausdruck, der ein Feld repräsentiert, gilt:

$$\text{code}_R e \rho = \text{code}_L e \rho$$

Analog zum Laden zusammengesetzter Strukturen benötigen wir auch die Möglichkeit, Strukturen abzuspeichern. Dazu verallgemeinern wir die Instruktion **store** zu Instruktionen **store** m für nicht-negative m (Abb. 2.19). Die allgemeine Form einer



Abb. 2.19. Die Instruktion **store** m .

Zuweisung eines Werts vom Verbundtyp t ergibt sich deshalb zu:

$$\begin{aligned} \text{code}_R (e_1 \leftarrow e_2) \rho &= \text{code}_R e_2 \rho \\ &\quad \text{code}_L e_1 \rho \\ &\quad \text{store } |t| \end{aligned}$$

Natürlich erlauben wir uns gegebenenfalls auch die Abkürzungen:

$$\begin{aligned} \text{loada } q \ m &= \text{loadc } q \\ &\quad \text{load } m \\ \text{storea } q \ m &= \text{loadc } q \\ &\quad \text{store } m \end{aligned}$$

Da wir nun auch nicht mehr davon ausgehen können, dass ein Ausdruck stets einen R-Wert der Größe 1 hat, müssen wir natürlich auch dafür sorgen, dass bei einer Anweisung e ; alle m Zellen vom Keller entfernt, die zum Wert von e gehören. Statt

einer einzelnen Instruktion **pop** könnten wir einfach m solche Instruktionen einfügen – oder als Optimierung uns auch eine neue Instruktion **pop** m spendieren. Die Implementierung dieser Instruktion überlassen wir dem Leser.

2.8 Zeiger und dynamische Speicherbelegung

Zeiger und dynamische Speicherbelegung für anonyme Objekte sind zwei eng verwandte Konzepte in imperativen Programmiersprachen. Bisher haben wir nur die Speicherbelegung für Objekte betrachtet, die durch eine Deklaration eingeführt werden. In der Deklaration wird ein Name für das Objekt angegeben, und diesem Namen wird eine statische (Relativ-) Adresse zugeordnet. Kennt eine Programmiersprache Zeiger, so kann man diese benutzen, um auf namenlose Objekte zuzugreifen; mithilfe von Zeigern kann man dynamisch wachsende und schrumpfende verkettete Strukturen realisieren, wobei die einzelnen Objekte nicht durch eine Deklaration, sondern durch die Ausführung einer entsprechenden Anweisung erzeugt werden. Die Semantik von C ist bezüglich der *Lebensdauer* dynamisch erzeugter Objekte nicht sehr präzise spezifiziert. Natürlich kann die Implementierung einer Programmiersprache schon vor dem Ende der Lebensdauer den durch ein solches Objekt belegten Speicher wieder freigeben, ohne gegen die Semantik zu verstoßen, wenn sichergestellt ist, daß es dem laufenden Programm nicht möglich ist, auf das Objekt noch zuzugreifen. Den Prozess der Freigabe von Speicher, der von unerreichbaren Objekten belegt ist, nennt man *Speicherbereinigung* (engl. Garbage Collection).

Wie in Abschnitt 2.1 kurz skizziert, wird im unteren Teil des Datenspeichers bei Funktionseintritt ein Datenbereich für den gesamten lokalen Speicherbedarf der Funktion (und für organisatorische Zwecke) angelegt und bei Verlassen der Funktion wieder freigegeben. Diese kellerartige Speicherbelegung und Freigabe passt nicht zu der Lebensdauer dynamisch erzeugter Objekte; auch eine Speicherbereinigung wird i.A. Speicher nicht kellerartig und nicht synchronisiert mit Funktionsaustritten wieder freigeben. Deshalb werden dynamisch erzeugte Objekte in einem Speicherbereich, genannt *Halde* (engl. Heap) am anderen Ende des Speichers untergebracht. Die Halde wächst bei dynamischer Anlage eines Objekts in Richtung auf den Keller zu (Abb. 2.20). Auf die unterste belegte Zelle der Halde zeigt ein neues Register

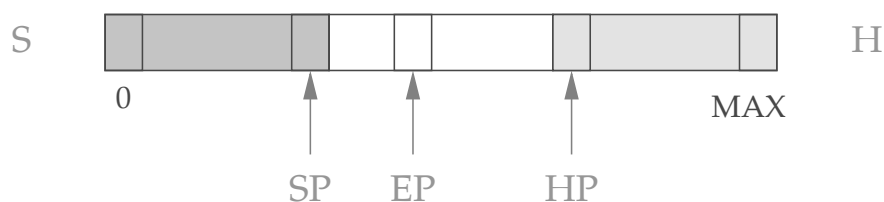


Abb. 2.20. Die Halde der C-Maschine.

der C-Maschine, der *Haldenzeiger* *HP* (Heap Pointer). Die Erzeugung eines neuen

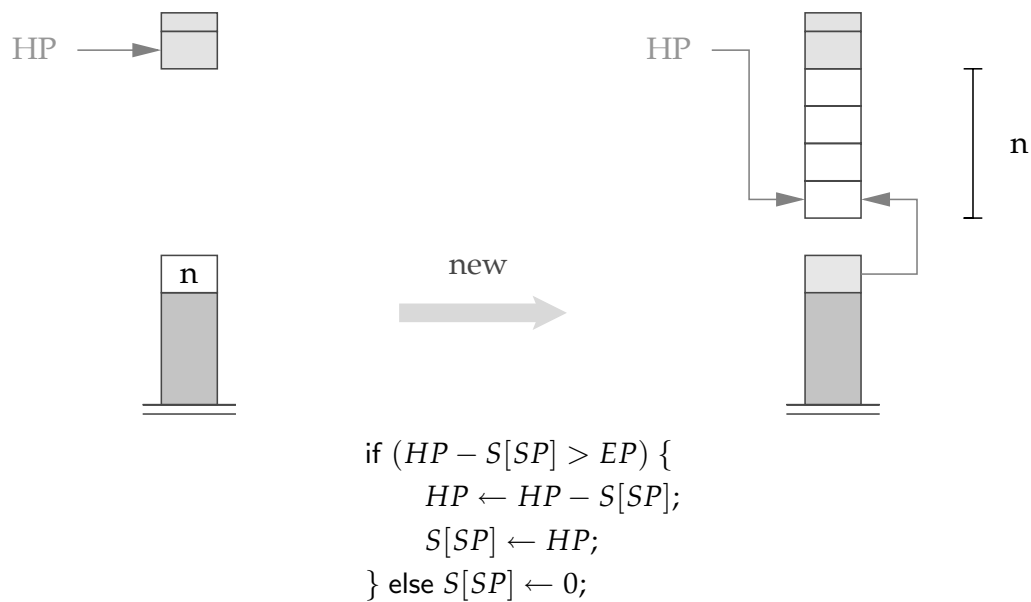


Abb. 2.21. Die Instruktion **new**.

Objekts auf der Halde geschieht mithilfe der Instruktion **new** (Abb. 2.21). Die Instruktion **new** erwartet oben auf dem Keller die Größe des zu erzeugenden Objekts und liefert die Anfangsadresse des für das Objekt zur Verfügung gestellten Speicherbereichs zurück. Vorher muss überprüft werden, ob noch genügend Speicherplatz für das neu anzulegende Objekt zur Verfügung steht. Ist nicht mehr genügend Platz vorhanden, liefert die Instruktion **new** den Wert 0 zurück.

Für die Überprüfung, ob Keller und Halde kollidieren, könnten wir einfach die Register *HP* und *SP* vergleichen. Dann müssten wir jedoch auch bei jeder Änderung der Kellerhöhe einen solchen Vergleich einfügen. Um diese (zahlreichen) Vergleiche einzusparen, verfügt unsere C-Maschine zusätzlich über das Register *EP* (Extreme Pointer) (Abb. 2.20). Dieses Register der C-Maschine soll dabei jeweils auf die oberste Kellerzelle zeigen, auf die *SP* bei der Auswertung von Ausdrücken im Anweisungsteil des aktuellen Funktionsaufrufs jemals zeigen kann. Deshalb bezeichnen wir *EP* auch als *Schrankenzeiger*. Wie in Übung 11 zu zeigen, lässt sich für jeden Ausdruck die maximale Anzahl der zu seiner Auswertung benötigten Kellerzellen zur Übersetzungszeit vorberechnen. Deshalb kann man den neuen Wert für *EP* bei Betreten einer Funktion aus dem *SP* ermitteln. Ein zu erwartender *Kellerüberlauf* (engl. Stack Overflow) wird damit bereits bei Betreten oder (wie wir später sehen werden) bei Verlassen eines Funktionsaufrufs festgestellt.

Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- Zeiger zu *erzeugen*, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu *dereferenzieren*, d.h. durch Zeiger auf die Werte von Speicherzellen zuzugreifen.

Technisch ist ein Zeiger nichts anderes als eine Speicheradresse. In C gibt es zwei Arten, Zeiger zu erzeugen: durch einen Aufruf der Bibliotheksfunktion **malloc** oder

durch die Anwendung des Adressoperators `&`. Ein Aufruf `malloc(e)` für einen Ausdruck e berechnet den R-Wert m von e und liefert einen Verweis auf die unterste Zelle eines neuen Speicherabschnitts der Größe m zurück. Mithilfe der Instruktion `new` können wir übersetzen:

$$\text{code}_R(\text{malloc}(e)) \rho = \text{code}_R e \rho$$

new

Beachten Sie, dass damit ein Aufruf der Funktion `malloc` niemals fehl schlägt. Auch wenn nicht mehr genügend Platz für das neue Objekt zur Verfügung steht, liefert der Aufruf eine Adresse zurück: im Fehlerfall jedoch die Adresse 0. Eine sorgfältige Programmiererin wird deshalb stets den Rückgabewert eines Aufrufs von `malloc` auf 0 testen, um diese Fehlersituation zu erkennen und angemessen behandeln zu können.

Die Anwendung `&e` des Adressoperators auf einen Ausdruck e liefert einen Zeiger auf das Speicherobjekt von e , d.h. auf dasjenige, das an der Anfangsadresse von e beginnt und den Typ von e besitzt. Folglich ist der R-Wert des Ausdrucks `&e` gerade der L-Wert des Ausdrucks e :

$$\text{code}_R(\&e) \rho = \text{code}_L e \rho$$

Sei e nun ein Ausdruck, der sich zu einem Zeigerwert p auswertet. Dieser Zeigerwert ist dann die Adresse des Objekts, auf das der Zeiger zeigt. Dieses Objekt erhalten wir, wenn wir den Zeiger p *dereferenzieren*, d.h. den Präfix-Operator `*` anwenden. Folglich stellt der R-Wert von e den L-Wert von `*e` dar. Das heißt, wir definieren:

$$\text{code}_L(*e) \rho = \text{code}_R e \rho$$

Eine Besonderheit der Programmiersprache C ist, dass sie *Zeigerarithmetik* unterstützt. Das bedeutet, dass man zu einem Zeigerwert p einen *int*-Wert a addieren oder von p den *int*-Wert a subtrahieren kann. Dieses Konzept soll unterstützen, mit einem Zeiger durch eine Folge gleichartiger Speicherobjekte zu wandern. Zeigt der Zeiger p auf einen Wert vom Typ t , dann bezeichnet der Ausdruck $p + a$ einen Zeiger auf das a -nächste Speicherobjekt. Für die Übersetzung heißt das, dass in einer Summe $e_1 + e_2$, in der e_1 vom Typ $t *$ und e_2 vom Typ **int** ist, der R-Wert von e_2 erst mit $|t|$ skaliert werden muss, bevor er zum R-Wert von e_1 addiert werden kann. Eine analoge Skalierung erfolgt ebenfalls bei der Subtraktion. Deshalb definieren wir in diesem Fall:

$$\begin{aligned} \text{code}_R(e_1 + e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{add} \\ \\ \text{code}_R(e_1 - e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{sub} \end{aligned}$$

Zur Übung leiten wir für einen Ausdruck e_1 vom Typ $t *$ und einen *int*-Ausdruck e_2 ein Übersetzungsschema für den Ausdruck $e_1[e_2]$ ab. Die Indizierung des Zeigerausdrucks ist eine Abkürzung für den Ausdruck $*(e_1 + e_2)$. Deshalb ergibt sich:

$$\begin{aligned} \text{code}_L e_1[e_2] \rho &= \text{code}_L (*(e_1 + e_2)) \rho \\ &= \text{code}_R(e_1 + e_2) \rho \\ &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{add} \end{aligned}$$

Interessanterweise entspricht dieses Schema unserem Schema für Feld-Indizierung. Bezeichnet der Ausdruck e_1 ein Feld, hatten wir zwar den L-Wert von e_1 benutzt und nicht den R-Wert wie im Falle von Verweisen. Für Felder stimmt der L-Wert jedoch mit dem R-Wert überein. Deshalb können wir das Übersetzungsschema für indizierte Zugriffe sowohl für Verweise wie Felder verwenden.

Zum Abschluss dieses Abschnitts betrachten wir ein etwas größeres Beispiel, an dem wir das Zusammenwirken der verschiedenen Übersetzungsschemata verfolgen können.

Beispiel 2.8.1 Für eine Deklaration:

```
struct  $t$  { int  $a[7]$ ; struct  $t *b$ ; };
int  $i, j$ ;
struct  $t *pt$ ;
```

wollen wir den Ausdruck $e \equiv ((pt \rightarrow b) \rightarrow a)[i + 1]$ übersetzen. Dabei ist der Operator \rightarrow eine Abkürzung für eine Dereferenzierung, gefolgt von einer Selektion. Das bedeutet:

$$e \rightarrow c \equiv (*e).c$$

Falls e den Typ $t *$ hat für eine Struktur t mit einer Komponente c , ergibt sich deshalb mit $m = \text{offsets}(t, c)$:

$$\begin{aligned}
 \text{code}_L(e \rightarrow c) \rho &= \text{code}_L((*e).c) \rho \\
 &= \text{code}_L(*e) \rho \\
 &\quad \text{loadc } m \\
 &\quad \text{add} \\
 &= \text{code}_R e \rho \\
 &\quad \text{loadc } m \\
 &\quad \text{add}
 \end{aligned}$$

In unserem Beispiel gilt $\text{offsets}(t, a) = 0$, und $\text{offsets}(t, b) = 7$. Nehmen wir an, wir hätten eine Speicherbelegungsfunktion ρ gegeben durch:

$$\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3\}$$

Dann ergibt sich für unseren Beispielausdruck e :

$$\begin{array}{llll}
 \text{code}_L e \rho & = & \text{code}_L((pt \rightarrow b) \rightarrow a) \rho & = & \text{code}_L((pt \rightarrow b) \rightarrow a) \rho \\
 & & \text{code}_R(i + 1) \rho & & \text{loada } 1 \\
 & & \text{loadc } 1 & & \text{loadc } 1 \\
 & & \text{mul} & & \text{add} \\
 & & \text{add} & & \text{loadc } 1 \\
 & & & & \text{mul} \\
 & & & & \text{add}
 \end{array}$$

Dabei ist:

$$\begin{array}{llll}
 \text{code}_L((pt \rightarrow b) \rightarrow a) \rho & = & \text{code}_R(pt \rightarrow b) \rho & = & \text{loada } 3 \\
 & & \text{loadc } 0 & & \text{loadc } 7 \\
 & & \text{add} & & \text{add} \\
 & & & & \text{load} \\
 & & & & \text{loadc } 0 \\
 & & & & \text{add}
 \end{array}$$

Insgesamt erhalten wir deshalb die Folge:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

Betrachten wir diese Instruktionsfolge, fällt uns zweierlei auf. Zum einen wären wir ohne unsere systematische Ableitung kaum so leicht auf diese Folge gekommen. Zum anderen erkennen wir aber auch, dass sie Raum für diverse Optimierungsmöglichkeiten bietet. So können wir die Addition von 0 einsparen, genauso wie natürlich die Multiplikation mit 1. Diese Ineffizienzen hätten wir direkt bei der Codeerzeugung vermeiden können, indem wir dort für die entsprechenden Sonderfälle eigene Übersetzungsschemata eingeführt hätten. Der Übersichtlichkeit halber haben wir auf diese Möglichkeit verzichtet und vertrauen stattdessen auf einen Postpass-Optimierer, der solche lokalen Codeverbesserungen gesondert durchführt. \square

Damit ist die Behandlung von Zeigern abgeschlossen. Bleibt nur noch die Frage, wie wir mit der eventuellen expliziten *Freigabe* von Speicherbereichen umgehen wollen. Die Freigabe des Speicherblocks, auf den ein Zeiger zeigt, ist jedoch nicht unproblematisch, weil es eventuell noch weitere Zeiger in diesen Speicherbereich gibt, sogenannte *hängende Zeiger* (engl. Dangling References).

Selbst wenn wir annehmen, dass die Programmiererin stets weiß, was sie tut, könnte nach einigen Freigaben die Halde so fragmentiert aussehen wie in Abb. 2.22. Die freien Speicherabschnitte können nach einiger Zeit sehr unregelmäßig über die

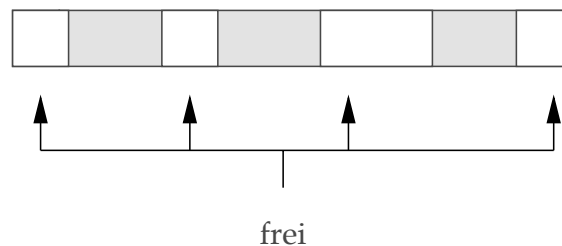


Abb. 2.22. Die Halde nach Freigabe einiger Blocks.

Halde verstreut sein. Sicherlich fallen Ihnen sofort eine Reihe von Techniken ein, wie Sie diese Abschnitte wieder einem Gebrauch durch das Programm zuführen könnten. In jedem Fall benötigt dazu Ihr Laufzeit-System weitere Datenstrukturen, deren Verwaltung Aufrufe der Funktionen **malloc** oder **free** verteuert.

In unserem minimalen Compiler verfolgen wir deshalb eine andere Strategie: bei einer Speicherfreigabe tun wir – nichts! Das ist sicherlich eine korrekte Implementierungsstrategie. Wenn sie auch nicht unbedingt speicheroptimal ist, so ist sie doch zumindest sehr einfach. Wir übersetzen also:

$$\text{code } (\mathbf{free}(e);) \rho \quad = \quad \text{code}_R e \rho \\ \mathbf{pop}$$

2.9 Funktionen

Zur Vorbereitung der Übersetzung von Funktionen wollen wir die zugehörigen Konzepte, Begriffe und Probleme kurz aufbereiten. So besteht die *Deklaration* einer Funktion aus:

- einem Namen, unter dem sie aufrufbar ist,
- der Spezifikation des Typs des Rückgabewerts,
- der Spezifikation der formalen Parameter, welche zusammen mit dem Rückgabewert die Ein-/Ausgabeschnittstelle bilden,
- einem *Rumpf*, welcher wiederum aus einer Folge von (lokalen) Deklarationen und Anweisungen besteht.

Falls die Funktion keinen Rückgabewert liefert, d.h. eine *Prozedur* ist, geben wir den Rückgabetyt als **void** an.

Funktionen werden *aufgerufen*, d.h. aktiviert, wenn ein Vorkommen ihres Namens im Anweisungsteil einer Funktion abgearbeitet wird. Eine aufgerufene Funktion kann ihrerseits wieder eine andere Funktion oder auch sich selbst aufrufen. Hat eine aufgerufene Funktion ihren Anweisungsteil vollständig abgearbeitet, so wird sie *verlassen* und ihr Aufrufer, d.h. die Funktion, die sie aktiviert hat, fährt in der Ausführung hinter dem Aufruf fort.

Betrachtet man die Funktionsaufrufe, die während der Ausführung eines Programms entstehen, so bilden sie einen geordneten Baum, den *Aufrufbaum* des Programmlaufs. Die Wurzel des Aufrufbaums ist markiert mit dem Namen der Funktion *main*, mit deren Aufruf die Programmausführung startet. Jeder innere Knoten im Aufrufbaum ist markiert mit einem Funktionsnamen f , sein direkter Vorgänger mit dem Namen der Funktion, welche diesen Aufruf von f ausgeführt hat; seine direkten Nachfolger bilden eine Liste von Funktionen, geordnet in der Reihenfolge ihrer Aufrufe durch f . Die Markierung f kann mehrfach im Aufrufbaum auftreten; genauer gesagt, tritt sie so oft auf, wie f im Laufe der Programmabarbeitung aufgerufen wird. Jedes Auftreten von f nennen wir eine *Inkarnation* von f . Eine Inkarnation ist charakterisiert durch den Weg von der Wurzel des Baums bis zu diesem Knoten. Diesen Weg nennen wir den *Inkarnationsweg* dieser Inkarnation von f .

Betrachten wir den Zustand der Programmausführung, wenn eine bestimmte Inkarnation von f aktiv ist. Nach dem oben Gesagten sind alle Vorfahren dieser Inkarnation, also alle Knoten auf dem Inkarnationsweg, bereits aufgerufen, aber noch nicht verlassen worden. Wir sagen, alle diese Inkarnationen sind zu diesem Zeitpunkt *lebendig*.

Beispiel 2.9.1 Betrachten wir das folgende C-Programm:

```

int n;
int fac(int n) {
    if (n ≤ 0) return 1;
    else return n * fac(n - 1);
}

int main() {
    int r;
    n ← 2;
    r ← fac(n) + fac(n - 1);
    return r;
}
```

Abbildung 2.23 zeigt den Aufrufbaum für dieses Programm. Der Pfeil zeigt auf den

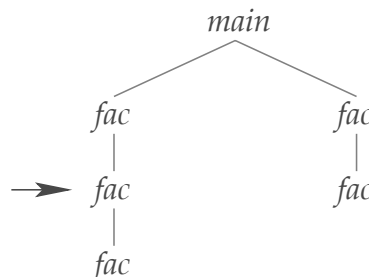


Abb. 2.23. Der Aufrufbaum für das Beispielprogramm.

zweiten rekursiven Aufruf der Funktion *fac*. Zu diesem Zeitpunkt sind offenbar die Hauptfunktion *main* und zwei rekursive Aufrufe von *fac* lebendig. □

Machen Sie sich an dieser Stelle klar,

- dass es zu einem Programm mehr als einen Aufrufbaum geben kann, und
- dass es auch unendliche Aufrufbäume gibt.

In einem Programm kann ein Name mehrmals vorkommen. Wir unterscheiden zwischen *definierenden Vorkommen* (Binding Occurrence) eines Namens, das sind diejenigen Vorkommen, bei denen ein Name in einer Deklaration definiert oder in einer formalen Parameterliste spezifiziert wird, und den andern, den sogenannten *angewandten Vorkommen* (Applied Occurrence).

Betrachten wir die in Funktionen auftretenden Namen. Die als formale Parameter oder durch lokale Deklarationen eingeführten Namen nennen wir *lokale Namen*. Im Abschnitt über funktionale Programmiersprachen werden sie uns als *gebundene Namen* wiederbegegnen. Wird eine Funktion aufgerufen, so werden für alle lokalen Namen neue *Inkarnationen* erzeugt. Dazu wird – entsprechend der Spezifikation bzw. Deklaration – Platz für die einfachen Variablen, Felder oder Verbunde bereitgestellt und (gegebenenfalls) mit einem Anfangswert belegt. Dies ist insbesondere für die formalen Parameter der Fall, die gemäß den aktuellen Parametern besetzt werden. Die *Lebensdauer* der erzeugten Inkarnationen ist gleich der Lebensdauer der Funktionsinkarnation. Der durch sie belegte Platz kann deshalb bei Verlassen der Funktion wieder freigegeben werden¹. Dies kann mit einer kellerartigen Speicher-verwaltung realisiert werden. Derselbe Speicherbereich, der bei Betreten der Funktion für die formalen Parameter, die lokal deklarierten Variablen und für anfallende Zwischenergebnisse (siehe Abb. 2.24) bereitgestellt wurde, wird bei Verlassen der Funktion auch wieder freigegeben.

Nicht ganz so einfach ist die Behandlung von angewandt auftretenden Namen, die nicht lokal sind. Wir bezeichnen diese Namen als *global* zur betrachteten Funktion. In funktionalen Programmen heißen entsprechende Vorkommen von Namen

¹ Wir ignorieren in dieser Einführung lokale Variablen, die gemäß einer Zusatzspezifikation (**own** in Algol60, **static** in PL/I oder C) ihre Funktionsinkarnationen überleben.

auch *frei*. Die *Sichtbarkeits-* und/oder *Gültigkeitsregeln* der Programmiersprache legen fest, wie das zu einem angewandten Auftreten eines Namens korrespondierende definierende Auftreten gefunden wird. Die umgekehrte, aber äquivalente Sicht geht von einem definierenden Vorkommen eines Namens aus und legt fest, in welchem Programmstück alle angewandten Vorkommen des Namens sich auf dieses definierende Vorkommen beziehen.

Aus Algol-ähnlichen Sprachen kennen wir folgende Sichtbarkeitsregel: Ein definierendes Auftreten eines Namens ist sichtbar in der Programmeinheit, in deren Deklarations- oder Spezifikationsteil die Definition steht, abzüglich aller von dieser Programmeinheit echt umfassten Programmeinheiten, die eine neue Definition des Namens enthalten. Dabei steht „Programmeinheit“ für eine Funktion oder einen *Block*. Blöcke werden in Aufgabe 12 näher betrachtet.

Ausgehend von dieser gegebenen Sichtbarkeitsregel wollen wir die beiden obigen Sichten noch einmal aufgreifen. Suchen wir zu einem angewandten Vorkommen das zugehörige definierende Vorkommen, so finden wir es, indem wir die Suche in dem Deklarationsteil der Programmeinheit beginnen, in der das angewandte Vorkommen steht. Ist keine solche vorhanden, dann setzen wir die Suche in der direkt umfassenden Programmeinheit fort usw. Findet sich in allen umfassenden Programmeinheiten einschließlich des gesamten Programms kein definierendes Vorkommen, so liegt ein Programmierfehler vor.

Die andere Sicht – ausgehend von einem definierenden Vorkommen – überstreicht gewissermaßen die enthaltende Programmeinheit und ordnet alle angetroffenen angewandten Vorkommen des Namens diesem definierenden Vorkommen zu. An den Grenzen von Programmeinheiten, die eine Neudefinition des gleichen Namens enthalten, wird der überstreichende Strahl abgeblockt.

Beispiel 2.9.2 Betrachten wir etwa das Programm aus Beispiel 2.9.1. Die Variable *n* ist außerhalb aller Funktionen definiert und damit insbesondere global zu der Funktion *main*. Da der formale Parameter der Funktion *fac* ebenfalls *n* heißt, ist diese globale Variable innerhalb des Rumpfs nicht sichtbar. Die angewandten Vorkommen von *n* innerhalb des Rumpfs beziehen sich damit stets auf den formalen Parameter der Funktion *fac*. □

Durch die so beschriebene Sichtbarkeitsregel erhält man die sogenannte *statische Bindung*, d.h. angewandt auftretende Namen, die global zu einer Programmeinheit sind, werden definierenden Vorkommen in textlich umgebenden Programmeinheiten zugeordnet. Diese Zuordnung ist statisch, da sie nur auf dem Programmtext und nicht auf der (dynamischen) Ausführung des Programms beruht. Jede Benutzung des globalen Namens zur Ausführungszeit betrifft eine Inkarnation des statisch zugeordneten definierenden Vorkommens.

Im Gegensatz dazu besteht die *dynamische Bindung* darin, dass ein Zugriff auf einen globalen Namen auf die zuletzt angelegte Inkarnation dieses Namens trifft – unabhängig davon, in welcher Funktion er definierend auftrat. Statische Bindung wird von allen Algol-ähnlichen Sprachen vorgeschrieben, aber auch in modernen funktionalen Sprachen wie Haskell oder OCaml, während einige ältere Lisp-Dialekte dynamische Bindung benutzen.

Beispiel 2.9.3 Betrachten Sie das folgende Programm:

```

int  $x \leftarrow 1$ ;
void  $q()$  {
     $\text{printf}("%d", x)$ ;
}

int  $\text{main}()$  {
    int  $x \leftarrow 2$ ;
     $q()$ ;
}

```

Bei statischer Bindung bezieht sich das angewandte Vorkommen der Variablen x in der Funktion q auf die globale Variable x und liefert deshalb den Wert 1. Das dynamisch zuletzt angelegte Vorkommen einer Variablen x vor dem Aufruf der Funktion q in der Funktion main ist dagegen die zu main lokale Variable x . Bei dynamischer Bindung sollte das angewandte Vorkommen von x in q deshalb den Wert 2 liefern.

□

Im Gegensatz zu Pascal erlaubt die Programmiersprache ANSI-C keine geschachtelten Funktionsdefinitionen. Diese Design-Entscheidung vereinfacht die Verwaltung der Sichtbarkeitsbereiche beträchtlich. Für ANSI-C genügt es, zwischen zwei Arten von Variablen zu unterscheiden: einerseits *globalen* Variablen, die außerhalb der Funktionsdefinitionen deklariert sind, und andererseits *lokalen* oder (im C-Jargon) *automatischen* Variablen, die lokal zu einzelnen Funktionen definiert sind.

2.9.1 Die Speicherorganisation der C-Maschine

Die im folgenden vorgestellte Speicherorganisation, der *Laufzeitkeller*, enthält zu jedem Zeitpunkt eine Folge von Speicherbereichen für die Menge der lebenden Inkarnationen, und zwar in der gleichen Reihenfolge, in der diese auf dem Inkarnationsweg auftreten. Um das Verlassen von Funktionen effizient zu realisieren, sind die Inkarnationen so miteinander verkettet, dass jede auf die Inkarnation zeigt, von der sie durch einen Aufruf angelegt wurde. Diese nennt man ihren *dynamischen Vorgänger*. Zu der Zelle für diesen Verweis kommen noch weitere organisatorische Zellen hinzu.

Im Speicher der C-Maschine wird für eine aufgerufene Funktion, d.h. die Inkarnation einer Funktion, ein *Kellerrahmen* (engl. Stack Frame) angelegt, dessen Organisation in Abb. 2.24 dargestellt ist. Insbesondere spendieren wir uns ein weiteres Register, den *Rahmenzeiger FP* oder *Frame Pointer*, der jeweils auf eine wohldefinierte Stelle des aktuellen Kellerrahmens zeigt. Im Kellerrahmen sehen wir Platz für die Register vor, deren Inhalt bei Betreten einer Funktion gerettet und bei Verlassen restauriert werden müssen. In der C-Maschine sind dies die Register *PC*, *FP* und *EP*. Der gerettete Inhalt des *PC* ist die *Rücksprungadresse*, an der die Berechnung nach Beendigung des Funktionsaufrufs fortfahren soll. Der gerettete Inhalt des *FP* ist der Verweis auf den Kellerrahmen der aufrufenden Funktion, d.h. den *dynamischen Vorgänger* des aktuellen Funktionsaufrufs. Dies ist der Anfang des aktuellen Inkarnationsweges im Aufrufbaum, auf dem alle lebenden Inkarnationen liegen. Der Inhalt des *EP* schließlich muss gerettet werden, weil er jeweils nur für den aktuellen Aufruf Gültigkeit besitzt.

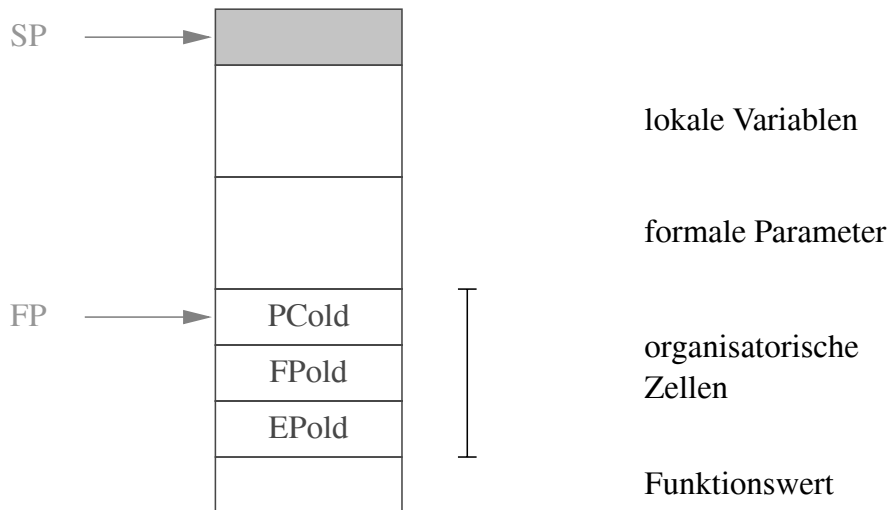


Abb. 2.24. Der Kellerrahmen der C-Maschine.

Die drei Zellen für die zu rettenden Register nennen wir auch die *organisatorischen Zellen*, weil mit ihrer Hilfe die Funktionsanfangs- und -endorganisation korrekt und effizient ablaufen.

Oberhalb der organisatorischen Zellen beginnt der Datenbereich der aktuellen Funktionsinkarnation. Hier legen wir konsequent die lokalen Variablen der aktuellen Funktion an. Das erlaubt uns, auf diese Variablen relativ zum Rahmenzeiger *FP* mit *festen Relativadressen* zuzugreifen. Zeigt der Rahmenzeiger auf die oberste organisatorische Zelle (d.h. den geretteten *PC*), können für die lokalen Variablen positive Relativadressen, angefangen mit 1 vergeben werden.

Oberhalb des Datenbereichs legen wir den *lokalen Keller*, das ist der Keller, der uns in Abschnitt 2.2 bei der Auswertung von Ausdrücken begegnet ist. Wie man sich überlegen kann, ist auch seine maximale Länge statisch bestimmbar (siehe Aufgabe 11).

Konzeptuell bleiben noch zwei Dinge, die im Kellerrahmen untergebracht werden sollten: das sind einerseits die aktuellen Parameter und andererseits der Rückgabewert der Funktion. Mit den aktuellen Parametern haben wir tatsächlich ein Problem: die Programmiersprache C erlaubt die Definition von Funktionen mit *variablen* Parameterlisten, wie etwa die Funktion *printf*, bei der der erste Parameter obligatorisch ist, aber die Anzahl der weiteren aktuellen Parameter erst aus dem Aufruf ersichtlich wird. Innerhalb einer solchen Funktion selbst kann also nur das Vorhandensein des Anfangsstücks der obligatorischen Parameter garantiert werden. Um für diese ebenfalls feste Relativadressen vergeben zu können, benutzen wir deshalb einen Trick: die aktuellen Parameter werden *unterhalb* der organisatorischen Zellen und in *umgekehrter Reihenfolge* auf den Keller gelegt! Das heißt, dass der erste Parameter oberhalb des zweiten zu liegen kommt usw. Als Relativadressen stehen nun

die *negativen Zahlen* ab -3 zur Verfügung. Hat der Typ des ersten Parameters z.B. die Größe m , bekommt er die Relativadresse $-(m - 2)$.

Liefert die Funktion einen Rückgabewert zurück, sollten wir für diesen einen kanonischen Platz vorsehen, auf den sich relativ zum FP mit einer festen Relativadresse zugreifen lässt. Wir könnten dazu etwa einen gesonderten Abschnitt unterhalb der organisatorischen Zellen wählen. Da aber bei Rückkehr aus einem Funktionsaufruf der Platz für die aktuellen Parameter mit Sicherheit nicht mehr benötigt wird, werden wir diesen Bereich zum Ablegen des Rückgabewerts wiederverwenden.

2.9.2 Der Umgang mit lokalen Variablen

In den Abschnitten 2.6 und 2.7 wurde beschrieben, wie man den in einem Deklarationssteil definierten Namen Speicherzellen bzw. Adressen zuordnen kann. Da wir dort nur Programme mit einem Deklarations- und einem Anweisungsteil, also ohne Funktionen betrachteten, gab es das Problem der Unterscheidung zwischen globalen und lokalen Variablen noch nicht. Tatsächlich verwendeten wir für die Variablenzugriffe *absolute Adressierung*, d.h. wir griffen auf Variablen relativ zur Anfangsadresse des Speichers zu. Die Adressumgebung ordnete dementsprechend den Namen ihre (Relativ-)Adressen zu.

Für ein richtiges C-Programm müssen wir uns einerseits vergegenwärtigen, dass ein Name auch eine Funktion bezeichnen kann. Auch diesen wollen wir Adressen zuordnen, nämlich die Anfangsadresse ihres Codes im Programmspeicher C .

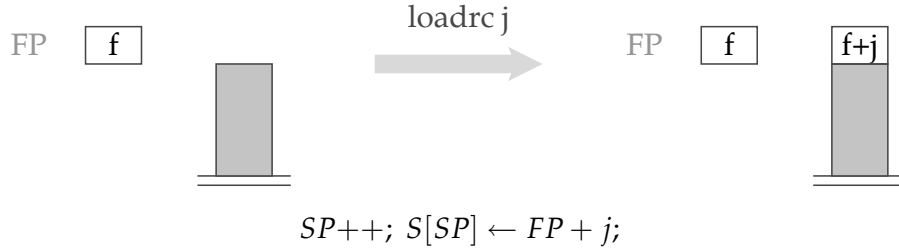
Bei formalen Parametern und lokalen Variablen dagegen wollen wir auf deren Instanzen im aktuellen Funktionsaufruf zugreifen. Die Adressierung erfolgt hier deshalb relativ zum Rahmenzeiger FP . Um diese Unterscheidung bei der Codeerzeugung berücksichtigen zu können, erweitern wir die Adressumgebung ρ so, dass ρ zu jedem definierenden Vorkommen eines Namens außer einer Relativadresse auch noch die Information verwaltet, ob der Name globale oder lokale Gültigkeit besitzt. Die Adressumgebung ρ hat deshalb jetzt die Funktionalität:

$$\rho : \text{Names} \rightarrow \{G, L\} \times \mathbb{Z}$$

wobei die Etiketten (engl. Tags) G und L jeweils globale bzw. lokale Gültigkeit bezeichnen. Um Zugriffe relativ zum FP auf lokale Variablen oder formale Parameter zu ermöglichen, reicht es, die Übersetzungsfunktion code_L für Namen zu verallgemeinern. Für $\rho(x) = (tag, j)$ definieren wir jetzt:

$$\text{code}_L x \rho = \begin{cases} \text{loadc } j & \text{für das Etikett } G \\ \text{loadrc } j & \text{für das Etikett } L \end{cases}$$

Dabei laden die neuen Befehle **loadrc** j den Wert $FP + j$ oben auf den Keller (Abb. 2.25). Selbstverständlich gestatten wir uns wieder als Optimierungen geeignete Spezialbefehle für häufig vorkommende Instruktionsfolgen:

Abb. 2.25. Die Instruktion **loadrc j**.

$$\begin{aligned} \mathbf{loadr\,j\,m} &= \mathbf{loadrc\,j} \\ &\quad \mathbf{load\,m} \\ \mathbf{storer\,j\,m} &= \mathbf{loadrc\,j} \\ &\quad \mathbf{store\,m} \end{aligned}$$

wobei wir auch **loadr j** und **storer j** für **loadr j 1** bzw. **storer j 1** schreiben.

Mit dieser Änderung können wir die Codeerzeugung, wie wir sie in den letzten Abschnitten Schritt für Schritt entwickelten, auch auf die Rümpfe von Funktionen anwenden.

Was wir aber zusätzlich benötigen, ist eine systematische Berechnungsmethode für die jeweils zu benutzende Adressumgebung ρ . Diese Berechnungsmethode muss dafür sorgen, dass an jeder Anwendungsstelle die Adressumgebung genau die dort sichtbaren Namen mit der jeweils aktuellen Information versieht.

Zur Bearbeitung einer globalen Variablendeklaration $t\,x$; definieren wir uns eine Funktion: **elab_global**, die für ein Paar (ρ, n) aus einer Adressumgebung ρ und einer ersten freien Relativadresse n sowie der Deklaration $d \equiv t\,x$ eine erweiterte Adressumgebung zusammen mit der nächsten freien Relativadresse liefert. Das heißt, wir definieren:

$$\mathbf{elab_global}(\rho, n)(d) = (\rho \oplus \{x \mapsto (G, n)\}, n + |t|)$$

Der Ausdruck $\rho \oplus \{x \mapsto a\}$ bezeichnet dabei die (partielle) Funktion, die man aus ρ erhält, indem man für das Argument x den Eintrag a hinzu fügt — bzw., falls ρ für x bereits definiert war, den alten Eintrag in ρ für x mit dem neuen Wert a überschreibt.

Analog benötigen wir Funktionen **elab_formal** und **elab_local** zur Bearbeitung von Deklarationen formaler Parameter bzw. lokaler Variablen x :

$$\begin{aligned} \mathbf{elab_formal}(\rho, z)(t\,x) &= (\rho \oplus \{x \mapsto (L, z - |t|)\}, z - |t|) \\ \mathbf{elab_local}(\rho, n)(t\,x) &= (\rho \oplus \{x \mapsto (L, n)\}, n + |t|) \end{aligned}$$

Die Funktion **elab_local** ist dabei völlig analog zur Funktion **elab_global** zur Behandlung der Deklaration einer globalen Variablen definiert — mit dem einen Unterschied, dass nun nicht das Etikett G , sondern das Etikett L vergeben wird. Bei der Definition der Funktion **elab_formal** müssen wir dagegen aufpassen, dass jeder weitere Parameter eine *kleinere* Adresse erhält. Anstelle der ersten freien Relativadresse

übergeben wir hier neben der Adressumgebung ρ die unterste bisher belegte Adresse z im Kellerrahmen.

Durch wiederholte Anwendung dieser Funktionen können wir ganze Listen von globalen Variablen, formalen Parametern bzw. lokalen Variablen abarbeiten:

$$\begin{aligned}
\text{elab_globals}(\rho, n) () &= (\rho, n) \\
\text{elab_globals}(\rho, n) (t\ x; ll) &= \text{elab_globals} (\text{elab_global} (\rho, n) (t\ x)) (ll) \\
\text{elab_formals} (\rho, z) () &= (\rho, z) \\
\text{elab_formals} (\rho, z) (t\ x, dd) &= \text{elab_formals} (\text{elab_formal} (\rho, z) (t\ x)) (dd) \\
\text{elab_locals}(\rho, n) () &= (\rho, n) \\
\text{elab_locals}(\rho, n) (t\ x; ll) &= \text{elab_locals} (\text{elab_local} (\rho, n) (t\ x)) (ll)
\end{aligned}$$

Nehmen wir an, wir hätten eine Funktion f ohne Rückgabewert mit einer Deklaration dd formaler Parameter und einer Deklaration ll lokaler Parameter. Aus einer Adressumgebung ρ für globale Namen erhalten wir damit als Adressumgebung ρ_f für die Funktion f :

$$\begin{aligned}
\rho_f = & \text{let } \rho = \rho \oplus \{f \mapsto (G, _f)\} \\
& \text{in let } (\rho, _) = \text{elab_formals} (\rho, -2) dd \\
& \text{in let } (\rho, _) = \text{elab_locals} (\rho, 1) ll \\
& \text{in } \rho
\end{aligned}$$

wobei $_f$ die Anfangsadresse des Codes von f bezeichne.

Besitzt die Funktion f einen Rückgabewert, verwalten wir die Relativadresse, an der der Rückgabewert im Kellerrahmen abgelegt wird, ebenfalls in der Adressumgebung für f . Dazu führen wir eine lokale Hilfsvariable `ret` ein. Sei t der Typ des Rückgabewerts und m der Platzbedarf der obligatorischen formalen Parameter.

- Ist $|t| \leq m$, können wir den Rückgabetyt am unteren Rand des Blocks für die formalen Parameter ablegen, d.h. ab Relativadresse $-(m+2)$. Dann erweitern wir unsere Definition von ρ_f um die Bindung $\text{ret} \mapsto (L, -(m+2))$.
- Ist $|t| > m$, benötigen wir eventuell einen größeren Block zur Rückgabe als wir für die Parameter benötigen. Dann legen wir den Rückgabewert ab Adresse $-(|t|+2)$ ab. Deshalb fügen wir dann zu ρ_f die Bindung $\text{ret} \mapsto (L, -(|t|+2))$ hinzu.

Beispiel 2.9.4 Betrachten wir erneut das C-Programm aus Beispiel 2.9.1. Als globale Namen haben wir n für eine globale Variable und fac und $main$ für die Funktionen. Dann ist

$$\rho_0 = \{n \mapsto (G, 1)\}$$

die Adressumgebung der vor der Deklaration der Funktion fac bekannten globalen Namen. Aus dieser Adressumgebung erhalten wir die Adressumgebung ρ_{fac} innerhalb von fac , indem wir zuerst die Bindung $fac \mapsto (G, _fac)$ zu ρ_0 hinzu fügen. Dann

müssen wir die Bindung $n \mapsto (L, 3)$ für den formalen Parameter n vermerken, welche die entsprechende Bindung für die globale Variable verdeckt. Schließlich finden wir, dass die Relativadresse des Rückgabewerts -3 ist. Wir erhalten also:

$$\rho_{fac} = \{n \mapsto (L, 3), \text{ret} \mapsto (L, -3), fac \mapsto (G, _fac)\}$$

Vor der Definition der Funktion *main* ist der Name der Funktion *fac* bereits bekannt. Sei deshalb

$$\rho_1 = \{n \mapsto (G, 1), fac \mapsto (G, _fac)\}$$

Dann ergibt sich die Adressumgebung ρ_{main} innerhalb der Funktion *main* als:

$$\rho_{main} = \{n \mapsto (G, 1), r \mapsto (L, 1), \text{ret} \mapsto (L, -3), \\ fac \mapsto (G, _fac), main \mapsto (G, _main)\}$$

Die Funktion *main* hat zwar keine formalen Parameter, dafür aber einen Rückgabewert der Größe 1. Deshalb erhält dieser wie bei der Funktion *fac* die Relativadresse -3 . Im Gegensatz zur Funktion *fac* wird innerhalb der Funktion *main* die globale Variable n nicht von einem gleichnamigen formalen Parameter verdeckt. Stattdessen enthält ρ_{main} eine Bindung für die zusätzliche lokale Variable r sowie eine Bindung für den globalen Funktionsnamen *main*. \square

2.9.3 Betreten und Verlassen von Funktionen

Betrachten wir die beiden entscheidenden Aufgaben bei der Implementierung von C-Funktionen, den Aufruf und damit den Eintritt in die Abarbeitung der Funktion und das Verlassen der Funktion nach Abarbeitung ihres Rumpfes.

Zunächst betrachten wir den Aufruf einer Funktion. Sei f die gegenwärtig aktive Funktion. Ihr Kellerrahmen ist damit der oberste auf dem Keller. Jetzt rufe die Funktion f eine Funktion g auf. Unser Ziel ist, für den Aufruf von g eine Instruktionsfolge zu generieren, die den Aufruf auswertet und den Rückgabewert oben auf dem Keller hinterlässt. Beachten Sie, dass unsere Instruktionsfolge damit einen R-Wert berechnet (sofern die Funktion nicht **void** zurückliefert). Insbesondere hat dieser Rückgabewert keinen (sinnvollen) L-Wert. Gemäß unseren Übersetzungsschemata heißt das, dass wir auf dem Rückgabewert einer Funktion nicht direkt eine Selektion ausführen können. Eine einfache Lösung dieses Problems besteht darin, vor der Codeerzeugung eine Programmtransformation anzuwenden, um die Rückgabewerte aller solcher problematischen Funktionsaufrufe in Hilfsvariablen abzulegen. Zum Beispiel würde eine Zuweisung $x \leftarrow f(y + 2).a$; von dieser Transformation übergeführt werden in den Block:

$$\{ \text{struct } t \text{ tmp} \leftarrow f(y + 2); x \leftarrow \text{tmp}.a; \}$$

für eine neue Variable *tmp*, wobei t der Rückgabotyp der Funktion f ist.

Die folgende Liste von Aktionen muss ausgeführt werden, um für einen Aufruf der Funktion g die Abarbeitung von g zu starten:

1. Die Werte der aktuellen Parameter müssen ermittelt und auf dem Keller hinterlassen werden.
2. Der alte Wert der Register *EP* und *FP* muss auf den Keller gerettet werden.
3. Die Anfangsadresse der Funktion *g* muss berechnet werden.
4. Die Rücksprungadresse muss ermittelt und in der entsprechenden organisatorischen Zelle vermerkt werden.
5. Das Register *FP* muss auf den aktuellen Kellerrahmen gesetzt werden.
6. Die Anfangsadresse von *g* muss angesprungen werden.
7. Das Register *EP* muss für den neuen Aufruf auf den aktuellen Wert gesetzt werden.
8. Für die lokalen Variablen von *g* muss auf dem Keller Platz reserviert werden.

Dann kann mit der Ausführung der Anweisungen des Rumpfs der Funktion *g* begonnen werden. Die Codeerzeugung muss diese Aktionen nun entsprechend dem jeweils vorhandenen Kenntnisstand auf den Aufrufer (engl. Caller) *f* und den Aufgerufenen (engl. Callee) *g* verteilen. So kann z.B. nur *f* die Werte der aktuellen Parameter ermitteln, und nur die Funktion *g* kennt den Platzbedarf für ihre lokalen Variablen. In unserer Liste verläuft die Grenze zwischen dem Code der aufrufenden und dem Code der aufgerufenen Funktion hinter dem Punkt (6). Für das Retten der Register *EP* und *FP* in Punkt (2) benutzen wir den neuen Befehl **mark** (Abb. 2.26). Dieser legt den Inhalt der beiden Register konsekutiv auf den Keller. Für das Anspringen der aufgerufenen Funktion in Punkt (6) benötigen wir den ebenfalls neuen Befehl **call** (Abb. 2.27). Er steht als letzter Befehl des Aufrufs von *g* im Aufrufer *f*. Bei Ausführung dieser Instruktion enthält das Register *PC* damit gerade die Rückkehradresse! Das kommt uns sehr gelegen, weil wir so die Aufgabe (5) von der Instruktion **call** miterledigen lassen können, indem diese einfach die Zieladresse ihres indirekten Sprungs oben auf dem Keller durch den aktuellen *PC* ersetzt. Der Einfachheit halber ordnen wir diesem Befehl auch noch die Aufgabe (4) zu, das Register *FP* auf den aktuellen Wert zu setzen, d.h. auf die oberste belegte Kellerzelle.

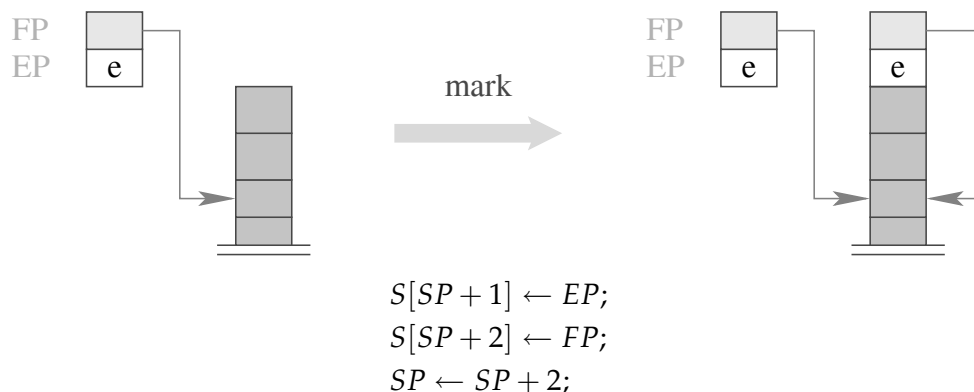
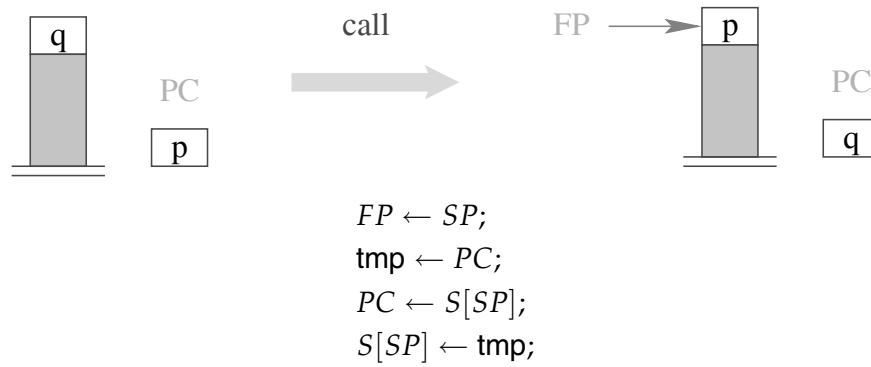
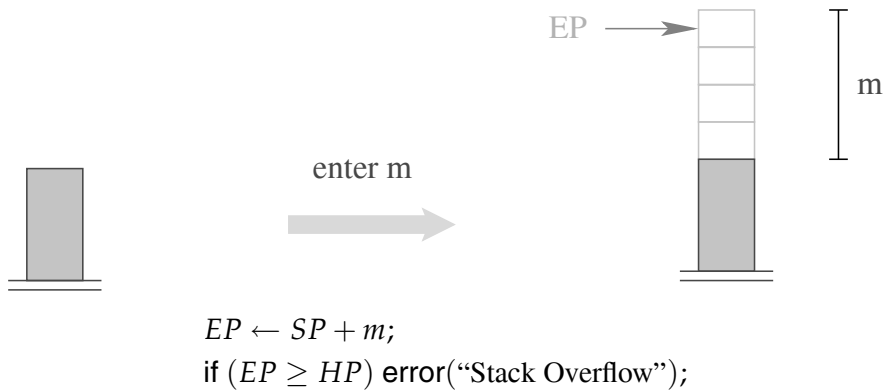


Abb. 2.26. Die Instruktion **mark**.

rufenen Funktion in Punkt (6) benötigen wir den ebenfalls neuen Befehl **call** (Abb. 2.27). Er steht als letzter Befehl des Aufrufs von *g* im Aufrufer *f*. Bei Ausführung dieser Instruktion enthält das Register *PC* damit gerade die Rückkehradresse! Das kommt uns sehr gelegen, weil wir so die Aufgabe (5) von der Instruktion **call** miterledigen lassen können, indem diese einfach die Zieladresse ihres indirekten Sprungs oben auf dem Keller durch den aktuellen *PC* ersetzt. Der Einfachheit halber ordnen wir diesem Befehl auch noch die Aufgabe (4) zu, das Register *FP* auf den aktuellen Wert zu setzen, d.h. auf die oberste belegte Kellerzelle.

Abb. 2.27. Die Instruktion **call**.

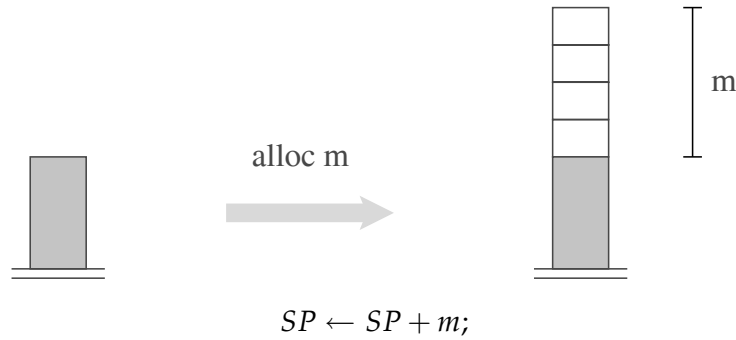
Damit verbleiben noch die Aufgaben (7) und (8). Den neuen *EP* setzen wir relativ zum aktuellen *SP* mithilfe eines Befehls **enter** *m*, wobei *m* die Gesamtanzahl der innerhalb der aufgerufenen Funktion benötigten Kellerzellen ist (Abb. 2.28). Es ist

Abb. 2.28. Die Instruktion **enter** *m*.

diese Instruktion, welche überprüft, ob auf dem Keller genügend Platz zur Abarbeitung des aktuellen Aufrufs vorhanden ist. Ist dies nicht der Fall, wird die Ausführung des Programms mit einer Fehlermeldung abgebrochen.

Zur Allokation von *m* Speicherzellen für die lokalen Variablen genügt es schließlich, das Register *SP* um *m* zu inkrementieren. Dafür führen wir die Befehle **alloc** *m* ein (Abb. 2.29). Dabei hat **alloc** 0 natürlich keinen Effekt.

Die *alloc*-Befehle können wir auch einsetzen, um auf dem Keller Platz für den Rückgabewert der Funktion zu allokalieren, falls der Platz für die aktuellen Parameter nicht ausreicht. Damit können wir bereits ein Übersetzungsschema für einen Funktionsaufruf *e* der Form $g(e_1, \dots, e_n)$ geben. Sei *t* der Rückgabebetyp von *g*, *m* der Platzbedarf für die formalen Parameter und $m' \geq m$ der Platzbedarf für die aktuellen Parameter. Dann haben wir:

Abb. 2.29. Die Instruktion **alloc** m .

```

codeR  $e \rho = \mathbf{alloc} \ q$ 
               codeR  $e_n \rho$ 
               ...
               codeR  $e_1 \rho$ 
               mark
               codeR  $g \rho$ 
               slide  $q' \ |t|$    where
                $q = \max(|t| - m', 0)$ 
                $q' = \text{if } (|t| \leq m) \ m' - m$ 
                   else  $\max(m' - |t|, 0)$ 

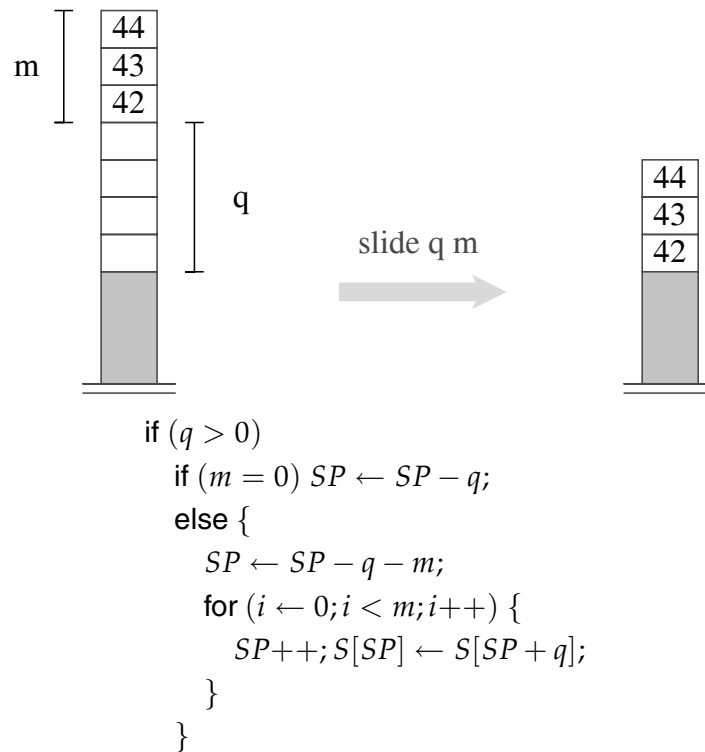
```

Zuerst allokatieren wir genügend Platz, um unterhalb der organisatorischen Zellen den Rückgabewert unterbringen zu können. Liegt der Rückgabewert dabei nicht am unteren Rand des Kellerrahmens, müssen wir ihn um entsprechend viele Zellen nach unten verschieben. Dazu dienen die Befehle **slide** $q \ m$ (Abb. 2.30).

Überzeugen Sie sich, dass gemäß unseres Übersetzungsschemas das erste Argument des zweiten *slide*-Befehls immer dann 0 ist, wenn die Funktion keine optionalen Parameter besitzt. Weiterhin sollten Sie überprüfen, dass niemals gleichzeitig eine Verschiebung des Ergebnisses nach dem Aufruf und ein Allokieren von zusätzlichem Speicher vor dem eigentlichen Aufruf erforderlich ist.

Sie sollten ebenfalls bemerkt haben, dass unser Schema für jeden der aktuellen Parameter e_i jeweils Code erzeugt, der den R-Wert von e_i in der Adressumgebung ρ berechnet. Dies entspricht einer Parameterübergabe an die Funktion *by value*. Andere imperative Programmiersprachen wie Pascal oder C++ unterstützen auch Parameterübergabe *by reference*. In diesem Fall sollte die Adresse, also der L-Wert des aktuellen Parameters an der entsprechenden Relativadresse des zugehörigen formalen Parameters x abgelegt werden. Jeder Zugriff auf den formalen Parameter x im Rumpf der Funktion erfordert dann eine *Indirektion* durch seine Relativadresse. Im Falle eines Referenzparameters x modifizieren wir deshalb die Berechnung des L-Werts von x zu:

$$\text{code}_L \ x \ \rho = \mathbf{loadr} \ \rho(x)$$

Abb. 2.30. Die Instruktion **slide** $q\ m$.

Nach dieser knappen Diskussion der Parameterübergabe möchten wir weiterhin darauf hinweisen, dass für den Ausdruck g , der die Anfangsadresse der aufzurufenden Funktion liefern soll, ebenfalls Code zur Berechnung des R-Werts erzeugt wird. Dies erlaubt insbesondere den Aufruf einer Funktion mithilfe eines Funktionszeigers. Einfache Funktionsnamen dagegen werden in C analog zu Feldern als *Referenzen* aufgefasst, deren R-Wert gleich dem L-Wert ist:

$$\text{code}_R f \rho = \text{code}_L f \rho = \text{loadc } \rho(f)$$

Beispiel 2.9.5 Betrachten wir den rekursiven Funktionsaufruf $fac(n-1)$ in dem Programm aus Beispiel 2.9.1. Dann liefert unser Übersetzungsschema die Instruktionsfolge:

alloc 0; loadr -3; loadc 1; sub; mark; loadc $_fac$; call; slide 0 1;

□

Nachdem wir eingehend die Aufgaben bei Betreten einer Funktion untersucht haben, wenden wir uns nun den Aufgaben zu, die bei Verlassen einer Funktion auszuführen sind. Diese sind:

1. (eventuell) Abspeichern des Rückgabewerts,
2. Rücksetzen der Register EP und FP ,
3. Aufräumen des Kellers und Rücksprung in den Code des Aufrufers.

Diese Aufgaben können offenbar sämtlich von der aufgerufenen Funktion ausgeführt werden. Da wir die Anfangsadresse des Rückgabewert – so es einen solchen gibt – in der Adressumgebung verwalten, kann die erste Aufgabe wie eine Zuweisung behandelt werden. Die anderen beiden Aufgaben fassen wir in der Instruktion **return** q zusammen (Abb. 2.31). Die Konstante q gibt dabei an, wie viele Zellen oberhalb des

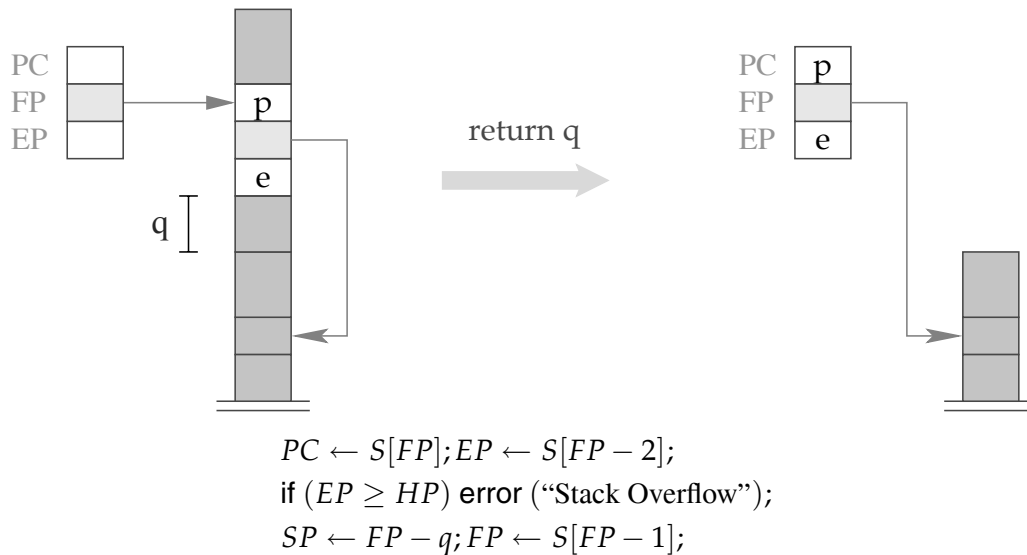


Abb. 2.31. Die Instruktion **return** q .

Rückgabewerts beseitigt werden sollen.

Beachten Sie, dass bei der Restauration des EP auf die Kollision von Keller und Halde getestet werden muss, da sich zwischenzeitlich der Haldenzeiger erniedrigt haben könnte. Ist der Wert des restaurierten EP nicht kleiner als der HP , bricht die Programmausführung mit einer Fehlermeldung ab.

Damit übersetzen wir die C-Anweisungen **return**; bzw. **return** e ; für einen Ausdruck e vom Typ t wie folgt:

`code (return;)` $\rho = \text{return } (m + 3)$
`code (return e ;)` $\rho = \text{code}_R e \rho$
 `storer` $\rho(\text{ret}) \mid t \mid$
 `return` q wobei
 $q = 3 + \max\{m - |t|, 0\}$

wobei m der Platzbedarf der Funktion für ihre formalen Parameter ist. Insgesamt übersetzen wir damit die Definition d einer Funktion f der Form:

$t \ f \ (params) \ \{ locals \ ss \}$

mit Rückgabebetyp t , Deklaration formaler Parameter $params$, Deklaration lokaler Variablen $locals$ und Anweisungsfolge ss wie folgt:

```
code  $d$   $\rho$  = enter  $k$ 
               alloc  $l$ 
               code  $ss$   $\rho_f$ 
               return  $q$ 
```

wobei k der maximale Platzbedarf bei einem Aufruf der Funktion f ist; l ist die Anzahl Speicherzellen für die lokalen Variablen, ρ_f ist die Adressumgebung für f , die man aus ρ , $params$ und $locals$ sowie der Größe des Rückgabetyps t erhält, und $q = \max\{m - |t|, 0\}$, falls m der Platzbedarf für die formalen Parameter ist.

Beispiel 2.9.6 Insgesamt können wir damit den Rumpf der Funktion fac aus Beispiel 2.9.1 so übersetzen:

$_fac$:	enter q	loadc 1	A:	loadr -3	mul
	loadr -3	storer -3		loadr -3	storer -3
	loadc 0	return 3		loadc 1	return 3
	leq	jump B		sub	
	jumpz A			mark	B : return 3
				loadc $_fac$	
				call	

Dabei haben wir die Instruktionen **alloc** 0 und **slide** 0 1 weggelassen. Auch beobachten wir, dass der Sprung mit Adresse B nicht erreichbar ist und damit ebenfalls wegoptimiert werden kann. \square

2.10 Übersetzung ganzer Programme

Als letztes beschreiben wir, wie ein gesamtes C-Programm p übersetzt wird. Dabei gehen wir davon aus, dass der Haldenzeiger HP vor der Programmausführung als Wert die größte Speicheradresse $+1$ hat und alle anderen Register den Wert 0 haben. Insbesondere heißt das, dass die Ausführung des CMA-Programms mit dem Befehl in $C[0]$ beginnt und SP den Wert 0 enthält.

Das Programm p ist eine Folge von Deklarationen von globalen Variablen und Funktionen, von denen eine die Funktion **int** $main()$ definiert. Beachten Sie, dass wir hier zur Vereinfachung keine Kommandozeilenparameter bei der Funktion $main$ erlauben. Der Code für das Programm p enthält damit:

- Code zum Anlegen der globalen Variablen;
- Code für die Funktionsdefinitionen;
- Code für den Aufruf von der Funktion $main$;

- die Instruktion **halt**, um die Programmausführung zu beenden.

Der Einfachheit halber nehmen wir an, dass die Variablendeklarationen in p vor den Funktionsdeklarationen stehen. Das Programm p ist also von der Form:

$$dd\ df_1 \dots df_n$$

wobei dd eine Folge von Variablendeklarationen ist und df_1, \dots, df_n jeweils die Deklarationen der Funktionen f_1, \dots, f_n sind mit $f_n \equiv \text{main}$. Bezeichne \emptyset die leere Adressumgebung. Dann ist

$$(\rho_0, k) = \text{elab_globals } (1, \emptyset) dd$$

das Paar aus der Variablenbelegung für die globalen Variablen von p und der ersten freien Relativadresse. Ist $_f_i$ die Anfangsadresse der Funktion f_i , dann ist die Adressumgebung vor Abarbeitung der i -ten Funktionsdefinition gegeben durch:

$$\rho_i = \rho_{i-1} \oplus \{f_i \mapsto _f_i\} \quad (i = 1, \dots, n)$$

Dann erhalten wir als Übersetzungsschema für p :

code p	=	enter $(k + 3)$
		alloc k
		mark
		loadc $_f_n$
		call
		slide $(k - 1) 1$
		halt
	$_f_1:$	code $df_1\ \rho_1$
		\vdots
	$_f_n:$	code $df_n\ \rho_n$

Vor dem Aufruf der Hauptfunktion f_n allokatieren wir oberhalb der Speicherzelle mit Adresse 0 insgesamt $(k - 1)$ Speicherzellen für globale Variablen sowie eine Zelle für den Rückgabewert von f_n . Da der SP vor der Ausführung den Wert 0 hat, müssen wir ihn um k erhöhen. Bis zur Ausführung der ersten Instruktion **call** werden insgesamt drei weitere Zellen auf den Keller gelegt. Deshalb muss am Anfang der *enter*-Befehl mit dem Argument $k + 3$ ausgestattet werden. Nach Abarbeitung des ersten Aufrufs liegt der Rückgabewert in der Speicherzelle mit Adresse k . Die Instruktion **halt**, die ihn an das Betriebssystem zurückliefern soll, erwartet ihn aber in der Speicherzelle 1. Deshalb müssen wir ihn nach dem Aufruf um $k - 1$ Zellen nach unten verschieben. Das leistet der Befehl **slide** $(k - 1) 1$.

Warnungen

In diesem jetzt abzuschließenden Kapitel wurde um der Verständlichkeit willen stark vereinfacht. Einige harte Realitäten wurden ignoriert. Insbesondere ist die Begrenzung durch die festzulegende Wortgröße problematisch.

Greifen wir etwa die Behandlung von Konstanten heraus. Wir haben Konstanten immer als Operanden in CMA-Befehlen untergebracht, in einigen Befehlen sogar zwei auf einmal.

Die zu unserer C-Maschine verwandte P-Maschine für Pascal differenziert etwa zwischen den Fällen, in denen die konstanten Operanden in den Befehl hineinpassen, und solchen, in denen das nicht der Fall ist. Dann werden die großen Konstanten in einer am oberen Speicherende, oberhalb der Halde liegenden Konstantentabelle abgelegt. In den Instruktionen stehen dann nur Verweise in diese Tabelle. Es gibt dann verschiedene Instruktionen für eine Operation, die sich darin unterscheiden, ob sie ihre Operanden im Befehl oder in der Konstantentabelle erwarten. Im letzteren Fall stehen Adressen im Befehl selbst. Der Übersetzer, bzw. im Falle des Züricher P4-Übersetzers ein nachgeschalteter Assembler, generiert dann abhängig von der Größe der Konstanten unterschiedliche Befehle und legt entsprechend die Konstanten in den Befehlen oder der Konstantentabelle ab.

Offene Fragen und Vorwärtsverweise

In diesem Kapitel wurde intuitiv erklärt, *was* die Übersetzung einer C-ähnlichen Sprache in die Sprache einer geeigneten virtuellen Maschine ist. Bei der geeigneten Leserin und dem geeigneten Leser sollte jetzt vielleicht der Wunsch entstanden sein, zu erfahren, *wie* die angegebenen Übersetzungsschemata realisiert werden.

- Die Schemata sind definiert über die syntaktische Struktur von Programmen. Sie setzen diese syntaktische Struktur bereits als bekannt voraus. Prinzipiell ist es zwar möglich, die Schemata als rekursive Funktionen aufzufassen, die die Struktur von Programmen erkennen und diese anschließend übersetzen. Allerdings wäre dies kein effizientes Verfahren; denn die Schablonen für die Ausdrücke oder auch die beiden bedingten Anweisungen wüssten lokal nicht immer, welches das richtige als nächstes anzuwendende Schema wäre. Sie müssten deshalb ausprobieren und eine falsche Auswahl später rückgängig machen. Deshalb werden im nachfolgenden Band in den Kapiteln über lexikalische und syntaktische Analyse effiziente Methoden zur Erkennung der Struktur von Programmen vorgestellt.
- Bei der Übersetzung von Wertzuweisungen und Ausdrücken wurden die Typen von Variablen und Ausdrücken benutzt. Die Art ihrer Berechnung wurde nicht angegeben. Das ist eine Teilaufgabe der semantischen Analyse, welche in dem entsprechenden Kapitel des nächsten Bands behandelt wird.
- In den Beispielen konnte man leicht sehen, dass die Verwendung der angegebenen Übersetzungsschemata nicht immer zu den bestmöglichen, d.h. kürzesten

CMa-Code-Befehlssequenzen führte. Die Frage, wie man Übersetzungsschemata angeben kann, welche zu besseren, vielleicht sogar zu optimalen Befehlssequenzen führen, wird uns im Kapitel über die Codeerzeugung beschäftigen. Entscheidend ist dabei die Verwendung auch nichtlokaler Information über dynamische, d.h. Laufzeiteigenschaften, zu deren Ermittlung man geeignete Methoden benötigt. Auch die Techniken zu solcher statischer Programmanalyse werden ausführlich zu diskutieren sein.

- Die Verwendung von virtuellen Maschinen hat das Übersetzungsproblem etwas entschärft. Es ergeben sich neue Probleme, wenn man Zielprogramme für reale Maschinen erzeugen und dabei deren Architektur möglichst gut nutzen will. Die Probleme bestehen etwa darin, die Registersätze der Zielmaschine optimal auszunutzen und unter den eventuell vielen möglichen Befehlssequenzen für Quellprogrammstücke die besten auszuwählen. Auch diese Probleme werden wir in einem späteren Band im Kapitel über die systematisch behandeln.

2.11 Aufgaben

1. *Codeerzeugung für Ausdrücke.*

Erzeugen Sie CMa-Code für die folgenden Ausdrücke:

$$\begin{aligned} e_1 &\equiv a \leftarrow 2 \cdot (c + (b - 3)) \\ e_2 &\equiv b \leftarrow b \cdot (a + 3) \end{aligned}$$

Benutzen Sie dazu die folgende Adressumgebung:

$$\rho = \{a \mapsto 5, b \mapsto 6, c \mapsto 7\}$$

Führen Sie den erzeugten Code aus!

Führen Sie den erzeugten Code aus, indem Sie den Stackzustand nach jedem Befehl angeben! Die Variablen seien mit den Werten $a = 22$, $b = 33$ und $c = 44$ initialisiert.

2. *Codeerzeugung für Schleifen.* Erzeugen Sie CMa-Code für die beiden Schleifen:

```

while ( $x > y$ ) {
    if ( $2 \cdot y > x$ )  $y \leftarrow y + x$ ;
    else  $x \leftarrow x - y$ ;
}

for ( $x \leftarrow 0; x < 42; x \leftarrow x + z$ )
    if ( $\neg(x = y)$ )  $z \leftarrow z + 1$ ;

```

Verwenden Sie dabei die Adressumgebung:

$$\rho = \{x \mapsto 2, y \mapsto 3, z \mapsto 5\}$$

3. *Codeerzeugung für Anweisungsfolgen.* Gegeben sei die Anweisungsfolge:

```

z ← 1;
while (n > 0) {
  j ← 1;
  y ← x;
  while (2 · j ≤ n) {
    y ← y · y;
    j ← j · 2;
  }
  z ← y · z;
  n ← n - j;
}

```

- Was berechnet die Anweisungsfolge?
- Erzeugen Sie CMa-Code für die Anweisungsfolge! Verwenden Sie dazu die Adressumgebung:

$$\rho = \{n \mapsto 1, j \mapsto 2, x \mapsto 3, y \mapsto 4, z \mapsto 5\}$$

4. *Reverse Engineering.* Gegeben sei der folgende CMa-Code:

loadc 0		pop	storea 1
loadc 1		jump B	pop
loadc 13	<i>A</i> :	loada 3	loadc 2
loada 3		loada 2	loada 2
loadc 1		geq	mul
le		jumpz B	storea 2
jumpz A		loada 1	pop
loadc -1		loadc 1	jump A
storea 1		add	<i>B</i> : halt

- Führen Sie den erzeugten Code aus. Gehen Sie davon aus, dass der *SP* vor der Programmausführung den Wert 0 hat.
 - Was berechnet der angegebene Code, wenn der Inhalt der Speicherzelle mit Adresse 3 als Eingabe betrachtet wird?
5. *Short circuit evaluation.* Seien b , e_1 und e_2 drei beliebige Ausdrücke.
- Ein bedingter Ausdruck in C hat die Form $b ? e_1 : e_2$. Sein Wert ist der Wert von e_1 falls $b \neq 0$ und e_2 falls $b = 0$.
Geben Sie ein Übersetzungsschema für $\text{code}_R(b ? e_1 : e_2) \rho$ an!
 - Unter *short circuit evaluation* für Boolesche Ausdrücke versteht man, dass das zweite Argument einer Konjunktion (bzw. Disjunktion) nicht mehr ausgewertet wird, wenn die Auswertung des ersten Arguments bereits 0 (bzw. einen Wert ungleich 0) liefert.

Geben Sie Übersetzungsschemata für $\text{code}_R(e_1 \wedge e_2) \rho$ (bzw. $\text{code}_R(e_1 \vee e_2) \rho$) an, die *short circuit evaluation* realisieren!

6. *Break.* Modifizieren Sie das Schema zur Übersetzung von Schleifen, so dass diese *break*-Anweisungen zum sofortigen Beenden der Schleife enthalten dürfen! Erweitern Sie dazu die Codeerzeugungsfunktion um ein weiteres Argument l , welches das Sprungziel angibt, zu dem bei einem **break** gesprungen werden soll!
7. *Continue.* Ein **continue** bewirkt einen Sprung an das Ende des Rumpfes der umfassenden Schleife. Wie muss man die Übersetzungsschemata abändern, damit **continues** an beliebiger Stelle richtig übersetzt werden?

Hinweis: Erweitern Sie die Übersetzungsfunktion um ein weiteres Argument!

8. *Sprungtabellen I.* Erweitern Sie das Übersetzungsschema für die *switch*-Anweisung so, dass in der Aufzählung der Fälle auch negative Werte bzw. (vereinzelte, kleine) Lücken vorkommen dürfen!

Entwickeln Sie Ideen, wie man auch mit sehr großen unregelmäßigen Lücken umgehen kann!

9. *Sprungtabellen II.* Die *switch*-Anweisung haben wir so übersetzt, dass anhand des Arguments ein relativer Sprung in die Sprungtabelle erfolgt, von wo ein weiterer (direkter) Sprung zu der Adresse des auszuführenden Codes führt. Den ersten Sprung kann man einsparen, wenn man in der Sprungtabelle anstelle von Sprungbefehlen die Sprungziele einträgt. Dann muss der Befehl **jumpi** durch einen Befehl **jumpi'** ersetzt werden, welcher direkt zu der in der Tabelle eingetragenen Adresse springt.

Dabei gibt es zwei Varianten:

- Die Sprungtabelle liegt *hinter* dem Code für die einzelnen Fälle. Dann bekommt **jumpi'** die Anfangsadresse der Tabelle als Argument;
- Die Sprungtabelle liegt unmittelbar hinter dem Befehl **jumpi'**, also *vor* dem Code für die einzelnen Fälle (solcher Code kann nur von einem Mehrpass-Compiler erzeugt werden). Dann entfällt die Anfangsadresse als Argument für den Befehl.

Definieren Sie für beide Fälle einen adäquaten virtuellen Befehl **jumpi'**, und geben Sie jeweils ein Übersetzungsschema an!

10. *Codeerzeugung für Zeiger.* Betrachten Sie die folgenden Definitionen:

```
int * m, n;
struct list {
    int info;
    struct list * next;
} * l, *tmp;
m ← malloc(sizeof(int));
*m ← 4;
l ← null;
...
```

```

...
for ( $n \leftarrow 5; n \leq *m; n--$ ) {
     $tmp \leftarrow l$ ;
     $l \leftarrow \text{malloc}(\text{sizeof}(\text{struct } list))$ ;
     $l \rightarrow next \leftarrow tmp$ ;
     $l \rightarrow info \leftarrow n$ ;
}

```

- Berechnen Sie eine Adressumgebung für die Variablen!
 - Erzeugen Sie Code für das Programm!
 - Führen Sie den erzeugten Code aus!
11. *Schrankenzeiger (Extreme Pointer)*. Zum Setzen des Schrankenzeigers *EP* während der Ausführung benötigt der Übersetzer den maximalen Zuwachs des Kellers (Stacks). Dieser Wert kann während der Übersetzung (statisch) berechnet werden.

Definieren Sie eine Funktion t , die einem Ausdruck e eine möglichst präzise obere Schranke $t(e)$ für die Anzahl der Kellerzellen zuordnet, die für die Auswertung des Ausdrucks e benötigt werden.

- Berechnen Sie $t(e)$ zunächst für die beiden Extremfälle:
 $a_1 + (a_2 + (\dots + (a_{n-1} + a_n) \dots))$ und $(\dots ((a_1 + a_2) + a_3) + \dots) + a_n$
 $(a_i \text{ Konstanten oder einfache Variablen}).$

- Berechnen Sie $t(e)$ für die in Aufgabe 1 sowie in Beispiel 2.8.1 angegebenen Ausdrücke.
 - Erweitern Sie den Definitionsbereich der Funktion t auf Anweisungen, insbesondere *if*-, *for*- und *while*-Anweisungen und Anweisungsfolgen.
12. *Blöcke*. Erweitern Sie die Codeerzeugungsfunktion für Anweisungsfolgen um Blöcke. Berücksichtigen Sie dabei, dass Variablendeklarationen an beliebigen Stellen innerhalb von Blöcken vorkommen dürfen. In Blöcken deklarierte Variablen sollen von dem Punkt, an dem sie eingeführt werden, bis zum zugehörigen Blockende sichtbar sein.
- Bei der Ausführung des folgenden Beispielsprogramms wird erst 2 und anschließend 1 als Wert der Variablen x ausgegeben.

```

int x;
 $x \leftarrow 1$ ;
{
    int x;
     $x \leftarrow 2$ ;
    write( $x$ );
}
write( $x$ );

```

Hinweis: Verwalten Sie zusätzlich zur Adressumgebung die erste vergebbare Relativadresse.

13. *Initialisierung von Variablen.* Modifizieren Sie die Übersetzungsfunktion `code` so, dass Variablen initialisiert werden können. Beispielsweise soll die Initialisierung:

int $a \leftarrow 0$;

übersetzt werden können.

14. *Post- und Pre-Inkrementierung.* Geben Sie Übersetzungsfunktionen an für:
- `++` und `--` (prefix und postfix).
 - `+ ←` und `- ←`.

Beachten Sie, dass die Operatoren nicht nur für Variablen definiert sind, sondern auf beliebige Ausdrücke angewendet werden können.

15. *Codeerzeugung für Funktionen.* Betrachten Sie die Definitionen:

```
int  $n$ ;
struct  $tree$  {
    int  $info$ ;
    struct  $tree$  *  $left$ , *  $right$ ;
} *  $t$ ;
struct  $tree$  *  $mktree$  (int  $d$ , int *  $n$ ) {
    struct  $tree$  *  $t$ ;
    if ( $d \leq 0$ ) return null;
    else {
         $t \leftarrow \text{malloc}(\text{sizeof}(\text{struct } tree));$ 
         $t \rightarrow left \leftarrow mktree(d - 1, n);$ 
         $t \rightarrow info \leftarrow *n$ ;  $*n \leftarrow *n + 1$ ;
         $t \rightarrow right \leftarrow mktree(d - 1, n);$ 
        return  $t$ ;
    }
}
```

Berechnen Sie Adressumgebungen für die Variablen (die globalen und die der Funktion) und erzeugen Sie Code für die Funktion und den Programmaufruf:

$t \leftarrow mktree(5, \&n)$;

Kommentieren Sie den erzeugten Code, so dass man seine Korrektheit nachvollziehen kann!

16. *Referenz-Parameter.* C++ bietet neben Parameterübergabe „by-value“ auch Referenz-Parameter an. Wird z.B. im Rumpf der Funktion f der formale Parameter x verwendet, ist die Variable gemeint, mit der f als aktuellem Parameter aufgerufen wurde.


```

int a;
void f(int &x) { x ← 7; }
int main() {
    f(a); return 0;
}

```

Nach Ausführung von $f(a)$ sollte also die Variable a den Wert 7 enthalten.

- Modifizieren Sie das Übersetzungsschema für Parameterübergabe
- und für Variablenzugriffe so, dass Referenz-Parameter korrekt behandelt werden!
- Übersetzen Sie das Programm und überprüfen Sie die Korrektheit!

17. *Variable Argumentlisten.* In dieser Aufgabe betrachten wir Funktionen mit variablen Argumentlisten. Bei solchen Funktionen steht am Ende der Parameter-Liste „...“ (davor können normale Parameter stehen). Innerhalb des Rumpfs der Funktion erhält man das jeweils nächste (variable) Argument von Typ t mit einem Aufruf $next(t)$. Beispiel:

```

int sum(int n, ...) {    // Summe von n Zahlen
    int result ← 0;
    while (n > 0) {
        result ← result + next(int);
        n ← n - 1;
    }
    return result;
}

```

Mögliche Aufrufe sind z.B. $sum(5, a, b, c, d, e)$ oder $sum(3, 1, 2, 3)$.

- Überprüfen Sie, ob unsere Übersetzung für solche Funktionsaufrufe geeignet ist!
- Überlegen Sie sich eine Übersetzung für $next(t)$!

18. *Codeerzeugung für ganze Programme.* Übersetzen Sie das Programm:

```

int result;
int fibonacci (int n) {
    int result;
    if (n < 0) return -1;
    switch (n) {
        case 0 : return 0; break;
        case 1 : return 1; break;
        default : return fibonacci (n - 1) + fibonacci (n - 2);
    }
}
...

```

```

...
int main() {
    int n;
    n ← 5;
    result ← fibonacci (n);
    return 0;
}

```

19. *CMa-Interpreter*. Implementieren Sie einen Interpreter für die CMa in der Programmiersprache Ihrer Wahl!

- Denken Sie sich einen geeigneten Datentyp für Instruktionen aus! Bedenken Sie dabei, dass manche Instruktionen ein Argument haben und manche nicht;
- Realisieren Sie die Datenstrukturen *C* und *S* durch Felder!
- Testen Sie Ihren Interpreter anhand der Fakultätsfunktion und dem Hauptprogramm:

```

int main() { return fac(9); }

```

2.12 Liste der Register der CMa

EP , Schrankenzeiger (Extreme Pointer)	S. 29
FP , Rahmenzeiger (Frame Pointer)	S. 37
HP , Haldenzeiger (Heap Pointer)	S. 28
PC , Befehlszähler (Program Counter)	S. 9
SP , Kellerzeiger (Stack Pointer)	S. 8

2.13 Liste der Codeerzeugungsfunktionen der CMa

code	S. 13
code_L	S. 13
code_R	S. 13

2.14 Liste der CMa-Instruktionen

add	S. 10	jumpz	S. 16	neg	S. 11
and	S. 10	jumpi	S. 20	neq	S. 11
alloc	S. 44	leq	S. 11	new	S. 29
call	S. 43	le	S. 11	or	S. 10
div	S. 10	load	S. 13	pop	S. 15
dup	S. 21	loada	S. 15	return	S. 47
enter	S. 44	loadc	S. 10	slide	S. 45
eq	S. 11	loadr	S. 40	store	S. 13
geq	S. 11	loadrc	S. 39	storea	S. 15
gr	S. 11	malloc	S. 29	storer	S. 40
halt	S. 49	mark	S. 43	sub	S. 10
jump	S. 16	mul	S. 10		

2.15 Literaturhinweise

Sprachorientierte virtuelle Maschinen gibt es schon recht lange. Sie wurden benutzt, um die Übersetzung zu vereinfachen und die Portierung auf andere Maschinen zu erleichtern. In [RR64] wurde eine virtuelle Maschine für Algol60, der *Algol Object Code* (AOC), beschrieben.

Vorbild für die heute üblichen virtuellen Maschinen für imperative Sprachen ist die P-Maschine, die für den weltweit verbreiteten Züricher P4-Übersetzer für Pascal verwendet wurde. Sie ist beschrieben in [Amm81] und in [PD82], worin sich die Quellen des P4-Übersetzers, des Assemblers und des P-Maschineninterpreters finden.

Seit einigen Jahren hat die *Java Virtual Machine* (JVM), die virtuelle Maschine für die objektorientierte Programmiersprache Java [LY99], der P-Maschine den Rang abgelassen.

Übersetzung funktionaler Programmiersprachen

3.1 Sprachtyp und einleitende Beispiele

Funktionale Programmiersprachen haben ihren Ursprung in Lisp, gehen also bis etwa 1958 zurück. Aber erst seit Ende der 70er Jahre hat sich diese Sprachklasse von der Dominanz von Lisp befreit und durch die Entwicklung neuer Konzepte und Implementierungsmethoden zu dem Sprachtyp entwickelt, den wir – repräsentiert durch OCaml hier behandeln wollen.

Imperative Sprachen kennen (mindestens) zwei Welten, die Welt der Ausdrücke und die Welt der Anweisungen. Ausdrücke liefern Werte; Anweisungen verändern den Zustand von Variablen oder steuern den Kontrollfluss. Bei entsprechender Programmierung kann auch bei der Auswertung von Ausdrücken als Seiteneffekt der Zustand von Variablen verändert werden. In funktionalen Sprachen gibt es (zuerst einmal) nur Ausdrücke. Die Ausführung eines funktionalen Programms besteht in der Auswertung des zugehörigen *Programmausdrucks*, der das Programmergebnis beschreibt. Zu seiner Auswertung müssen evtl. weitere Ausdrücke ausgewertet werden, etwa wenn ein Ausdruck Funktionsanwendungen enthält. Es gibt keinen explizit durch Anweisungen angegebenen Kontrollfluss. Variablen in funktionalen Programmen können an Ausdrücke *gebunden* werden. Variablen sind deshalb nicht Bezeichner für Speicherzellen, deren Wert wie in imperativen Sprachen geändert werden kann. Die Ausführung des Programms kann lediglich den an eine Variable gebundenen Ausdruck *auswerten*. Anschließend ist die Variable an dessen Wert gebunden.

Stellen wir kurz die wichtigsten Konzepte moderner funktionaler Programmiersprachen zusammen. In der konkreten Syntax lehnen wir uns dabei an die Syntax der Programmiersprache OCaml an:

Funktionsdefinitionen. Dies geschieht durch *Abstraktion*. Ein Ausdruck

$$\mathbf{fun} \ x \ y \rightarrow x + y$$

definiert eine Funktion mit den formalen Parametern x, y und dem definierenden Ausdruck $x + y$, der den Rückgabewert liefern soll. Damit hat man eine *anonyme*, d.h. namenlose Funktion definiert, die man auf zwei Argumente anwenden kann:

$$(\mathbf{fun} \ x \ y \rightarrow x + y) \ 1 \ 2$$

Oft ist es praktisch, für eine durch Abstraktion definierte Funktion einen Namen einzuführen, den man anstelle der Funktion verwenden kann:

$$\begin{aligned} &\mathbf{let} \ add = \mathbf{fun} \ x \ y \rightarrow x + y \\ &\mathbf{in} \ add \ 1 \ (add \ 2 \ 3) \end{aligned}$$

Rekursionsgleichungen. Verwendet man den Namen einer Funktion im eigenen definierenden Ausdruck, hat man eine *rekursive Funktion* definiert. Technisch kann man eine solche Definition als (funktionale) *Rekursionsgleichung* auffassen. Die Definition

$$\begin{aligned} \mathbf{let} \ \mathbf{rec} \ log = \mathbf{fun} \ x \rightarrow &\mathbf{if} \ x \leq 1 \ \mathbf{then} \ 0 \\ &\mathbf{else} \ 1 + log \ (x/2) \end{aligned}$$

bindet den Namen *log* an den Ausdruck auf der rechten Seite, welcher selbst wieder auf den Namen der linken Seite der Definition Bezug nimmt. In Programmiersprachen mit verzögerter Auswertung sind rekursive Definitionen auch nicht-funktionaler Werte wie z.B. (potentiell) unendliche Listen möglich.

Pattern Matching. Zur Behandlung strukturierter algebraischer Datentypen wie z.B. Listen bieten funktionale Sprachen eine erweiterte Form von Fallunterscheidung an, bei der Werte mit *Mustern* (Patterns) verglichen werden können. Passt das Muster auf den Wert, werden die Variablen, die in dem Muster vorkommen, an die entsprechenden Komponenten des Werts gebunden. In dem Ausdruck:

$$\begin{aligned} &\mathbf{match} \ [1; 2; 3] \ \mathbf{with} \\ &\quad [1; x; y] \rightarrow x + y \\ &\quad | \quad [z] \rightarrow z \end{aligned}$$

wird z.B. die erste Alternative ausgewählt. Dann werden die Variablen *x* und *y* an die Komponenten 2 und 3 gebunden und deren Summe zurückgeliefert. Dieser Vergleich mit Mustern heißt auch *Pattern Matching*.

Höhere Funktionen. *Höhere Funktionen*, auch *Funktionen höherer Ordnung* genannt, sind Funktionen, die Funktionen als Argumente und/oder als Ergebnisse haben. Zur Erläuterung dieses Konzepts betrachten wir die in OCaml vordefinierte Funktion *map*. Die Funktion *map* bekommt als erstes Argument eine Funktion *f* und als zweites eine Liste *l*; die Listenelemente sind vom Typ der Argumente von *f*. Der Aufruf *map f l* wendet *f* auf alle Listenelemente in *l* an und gibt die Liste der Ergebnisse zurück:

$$\begin{aligned} \mathbf{let} \ \mathbf{rec} \ map = \mathbf{fun} \ f \ l \rightarrow &\mathbf{match} \ l \ \mathbf{with} \\ &[] \rightarrow [] \\ &| \ x :: xs \rightarrow f \ x :: map \ f \ xs \end{aligned}$$

Abhängig von dem Wert des zweiten Parameters unterscheidet die Funktion zwei Fälle. Wird sie auf eine Funktion und eine leere Liste angewandt, gibt sie eine leere

Liste zurück. Ist das zweite Argument dagegen eine nichtleere Liste, wendet sie die Funktion f im ersten Argument auf das erste Element der Liste an und konkateniert das Ergebnis mit der Liste, die sich aus der Anwendung von map auf f und den Rest der Liste ergibt. Die Funktion map ist eine *höhere* Funktion, weil sie selbst wieder eine Funktion als Argument erwartet.

Lassen Sie uns hier bemerken, dass neben der bisher verwendeten puristischen Syntax die Programmiersprache OCaml auch erlaubt, die formalen Parameter in einer Funktionsdefinition hinter dem Namen auf der linken Seite aufzulisten:

```
let rec map f l = match l with
  [] → []
  | x :: xs → f x :: map f xs
```

Zur Vereinfachung der Definition von Funktionen, die über Muster in einem Argument definiert werden, bietet OCaml hier sogar noch eine Spezialsyntax an:

```
let rec map f = function
  [] → []
  | x :: xs → f x :: map f xs
```

Polymorphie. Typen in funktionalen Programmiersprachen werden meist als Terme dargestellt. Basistypen sind atomare Terme, wie **int**, **bool**. Strukturierte Typen wie Listen werden durch Typterme beschrieben, ein Liste von **int**-Werten etwa durch **int list**. Der Typ eines Ausdrucks kann aber auch durch ein *Typschema* gegeben sein. Ein Typschema enthält dabei *Typvariablen*, die bei verschiedenen Benutzungen mit unterschiedlichen Typen instantiiert werden können. Dieses Typschema kann durch eine Deklaration angegeben sein oder von einem *Typinferenzalgorithmus* aus den Anwendungsstellen der Funktion abgeleitet sein. Ein Typinferenzalgorithmus leitet für alle Größen, für die die Programmiererin keine Typen angegeben hat, ein möglichst allgemeines (oft *das* allgemeinste) Typschema her und prüft, ob alle Ausdrücke typkorrekt aufgebaut sind.

Die eben vorgestellte Funktion hat den *polymorphen* Typ:

$$map : ('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$$

Die Typvariablen $'a$, $'b$ stehen für beliebige Typen. Die Funktion map können wir damit auf eine *int*-Funktion und eine *int*-Liste genauso anwenden wie auf eine Boolesche Funktion und eine Liste Boolescher Werte. Der Typ von map verlangt nur, dass der Argumenttyp des ersten, funktionalen Arguments mit dem Typ der Elemente des zweiten Arguments, einer Liste, übereinstimmt. Das Ergebnis hat dann einen Listentyp, wobei die Listenelemente den Typ des Ergebnisses des funktionalen Arguments haben.

In den folgenden Abschnitten werden wir schrittweise die Architektur und den Befehlsvorrat der virtuellen Maschine MaMa und die Übersetzungsschemata für die Übersetzung von FuL in MaMa-Code angeben. Nach einer Schilderung der Architektur werden wir wie in Abschnitt 2.2 mit der Übersetzung von Ausdrücken beginnen. Bis auf die Behandlung von Variablen gehen wir ganz ähnlich vor wie bei der

Übersetzung von C in C-Maschinencode. Neue Konzepte dagegen sind erforderlich, um verzögerte Auswertung und höhere Funktionen zu realisieren.

3.2 Eine einfache funktionale Programmiersprache

Wir führen eine einfache funktionale Programmiersprache ein, an der wir das Prinzip der Übersetzung funktionaler Programmiersprachen erklären. Dabei interessiert uns – wie im Kapitel 2 – mehr eine geeignete virtuelle Maschine und die Codeerzeugung für eine solche als die sonstigen Aufgaben der Übersetzung wie etwa die Typüberprüfung. Zur Vereinfachung betrachten wir einen kleinen Ausschnitt aus der Programmiersprache OCaml, den wir FuL (**F**unctional **L**anguage) nennen wollen. Zur weiteren Vereinfachung erlauben wir im Unterschied zu OCaml als einzigen Basistyp **int**. Das heißt, wir nehmen wie bei der Programmiersprache C an, dass Bedingungen ganze Zahlen zurückliefern, welche wir auf null (**false**) oder nicht-null (**true**) testen können. Später werden wir unsere Kernsprache um einfache Datenstrukturen wie Tupel und Listen erweitern.

Ein Programm ist damit zuerst einmal ein Ausdruck e der Form:

$$\begin{aligned} e ::= & b \mid x \mid (\square_1 e) \mid (e_1 \square_2 e_2) \\ & \mid (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \\ & \mid (e' e_0 \dots e_{k-1}) \mid (\text{fun } x_0 \dots x_{k-1} \rightarrow e) \\ & \mid (\text{let } x = e \text{ in } e_0) \\ & \mid (\text{let rec } x_1 = e_1 \text{ and } \dots \text{ and } x_n = e_n \text{ in } e_0) \end{aligned}$$

wobei \square_1 und \square_2 beliebige einstellige bzw. zweistellige Operatoren auf **int**-Werten bezeichnen. Ein Ausdruck ist somit:

- ein Basiswert, eine Variable, eine Operator-Anwendung oder ein bedingter Ausdruck;
- eine Funktionsanwendung;
- eine Funktion – d.h. aus einem Funktionsrumpf entstanden mithilfe von Abstraktion der formalen Parameter;
- ein *let*-Ausdruck, der eine lokale Definition einführt, oder
- ein *letrec*-Ausdruck, der lokal rekursive Definitionen einführt.

Beispiel 3.2.1 Die folgende allseits bekannte Funktion berechnet die Fakultät (und insbesondere den Wert auf dem Argument 13):

$$\begin{aligned} \text{let rec } fac &= \text{fun } x \rightarrow \text{if } x \leq 1 \text{ then } 1 \\ &\quad \text{else } x \cdot fac (x - 1) \\ \text{in } fac &13 \end{aligned}$$

Wie üblich setzen wir nur da Klammern, wo sie zum Verständnis erforderlich sind, d.h. in einer ungeklammerten Folge von Ausdrücken $e_0 e_1 \dots e_k$ nehmen wir an, dass e_0 die Funktion repräsentiert und die Ausdrücke e_1, \dots, e_k die aktuellen Parameter der Funktion sind. \square

Die Semantik von Funktionsanwendungen benötigt noch zwei Festlegungen. Die erste betrifft die Parameterübergabe, nämlich was bei einer Funktionsanwendung an die Funktion übergeben wird. Die zweite betrifft die Interpretation *freier* Variablen. Diese hießen im letzten Kapitel *globale Variablen*. Für freie Variablen können wir statische oder dynamische Bindung verwenden, d.h. einer freien Variable durch ihre textuell innerste umgebende Bindung ihren Wert geben oder durch die zeitlich zuletzt hergestellte Bindung.

Der Parameterübergabemechanismus der Programmiersprache regelt, in welcher Form die aktuellen Parameter übergeben werden. Algol60 und nachfolgend Pascal und C kennen die *call-by-value* Übergabe, Algol60 außerdem noch *call-by-name* und Pascal wie auch C++ *call-by-reference*. Letztere verliert in funktionalen Programmiersprachen ihren Sinn, da diese keine Adressen kennen. Parameter können deshalb nur als Werte oder als *Ausdrücke* übergeben werden.

Ob in einer Funktionsanwendung $e_0\ e_1 \dots e_k$ die aktuellen Parameter e_1, \dots, e_k ausgewertet oder deren Auswertung nach Möglichkeit verzögert wird, kann durch die *Auswertungsordnung* charakterisiert werden: entweder werden zuerst die aktuellen Parameter e_1, \dots, e_k ausgewertet und anschließend mit der Auswertung der Funktion e_0 begonnen. Oder die Auswertung beginnt mit der Auswertung des Funktionsausdrucks e_0 zu einer Funktion **fun** $x_1 \dots x_m \rightarrow e$. Sind genügend Argumente da, d.h. ist $m \leq k$, werden die formalen Parameter x_i an die unausgewerteten Argumente e_i gebunden und mit der Auswertung des Rumpfs e fortgefahren. Die Auswertung der aktuellen Parameter e_i dagegen wird solange verzögert, bis deren Wert bei der Auswertung von e tatsächlich benötigt wird.

Beispiel 3.2.2 Betrachten Sie das funktionale Programm:

```

let rec   fac    =    ...
           and   foo    =    fun x y  $\rightarrow$  x
           in foo 1 (fac 1000)

```

wobei die Funktion *fac* die Fakultätsfunktion ist. Die Funktion *foo* soll den Wert des ersten Parameters zurückliefern. Die Reihenfolge der Abarbeitung macht hier einen großen Unterschied. Wird zuerst der aktuelle Parameter ausgewertet, muss eine relativ teure Berechnung des Werts des Teilausdrucks (*fac* 1000) durchgeführt werden, die sogar zu einer Bereichsüberschreitung mit nachfolgendem Programmabbruch führen kann. Der zweite Parameter ist jedoch zur Berechnung des Ergebnisses gar nicht erforderlich! Wird mit der Auswertung des Rumpfs der Funktion *foo* begonnen, terminiert die Auswertung sehr schnell und unter Umgehung einer eventuellen Bereichsüberschreitung und liefert den Wert 1. \square

Entsprechend diesen Überlegungen unterscheiden wir drei Fälle:

Call-By-Value oder *Applicative Order Evaluation*:

Die Parameter werden zuerst ausgewertet und ihre Werte an die Funktion übergeben. Das ist die Auswertungsstrategie etwa von OCaml.

Vorteil: Die Parameter werden nur einmal ausgewertet; kein zusätzlicher Aufwand entsteht außer der Auswertung.

Nachteil: Mancher Parameter wird ausgewertet, obwohl die Auswertung des Rumpfs den Wert dieses Parameters gar nicht benötigt. Das ist kritisch, falls die Auswertung des Parameters sehr teuer ist, zu einem Laufzeitfehler führt oder nicht terminiert.

Call-By-Name oder Normal Order Evaluation:

Es wird mit der Auswertung des Rumpfs einer Funktion begonnen; jedesmal, wenn der Wert eines Parameters benötigt wird, wird der zugehörige Ausdruck ausgewertet.

Vorteil: Ein Parameter wird nur ausgewertet, wenn sein Wert auch tatsächlich benötigt wird. In unserem Beispiel 3.2.2 würde damit die Auswertung des Aufrufs (*fac* 1000) vermieden. Allgemein gilt, dass *call-by-name* bessere Terminierungseigenschaften als *call-by-value* hat.

Nachteil: Wird der Wert eines Parameters mehrmals benötigt, wird er auch mehrmals ausgewertet.

Call-By-Need oder Lazy Evaluation:

Ein Parameter wird nur ausgewertet, wenn sein Wert benötigt wird, und dann nur einmal. Der erste Zugriff erzwingt also die Auswertung des Parameters. Alle weiteren Zugriffe greifen auf den bereits berechneten Wert zu. Diese Strategie wird z.B. von der Programmiersprache Haskell umgesetzt.

Call-by-need kann als eine Optimierung der Auswertungsstrategie *call-by-name* aufgefasst werden, die versucht, das gute Terminierungsverhalten von *call-by-name* mit der Effizienz von *call-by-value* zu vereinen. Es sollte jedoch nicht verschwiegen werden, dass eine Verzögerung der Auswertung von Teilausdrücken nicht zum Nulltarif zu haben ist, sondern zusätzlichen Verwaltungsaufwand erfordert. Für die Programmiersprache FuL werden wir die Frage der Parameterübergabe offen halten und bei der Übersetzung Schemata sowohl für *call-by-value* (CBV) wie für *call-by-need* (CBN) angeben.

Bei der Diskussion der Parameterübergabe haben wir bisher die Entscheidung zwischen statischer oder dynamischer Bindung offen gelassen, obwohl diese beiden Festlegungen nicht unabhängig voneinander sind. Bei *statischer Bindung* (Static Scoping) bezieht sich ein angewandtes Auftreten eines Namens immer auf das textuell innerste umgebende Konstrukt, welches diesen Namen definiert. Bei *dynamischer Bindung* (Dynamic Scoping) legt die dynamisch zuletzt für diesen Namen hergestellte Bindung den Wert fest.

Beispiel 3.2.3 Betrachten Sie das folgende kleine Programm:

```

let     $x = 2$  in
let     $f = \text{fun } y \rightarrow x + y$  in
let     $h = \text{fun } g\ x \rightarrow g\ 2$  in
         $h\ f\ 1$ 

```

Bei statischer Bindung bezieht sich die freie Variable x im Rumpf von f auf die Definition $x = 2$; dementsprechend ergibt sich für $h\ f\ 1$ als Wert 4. Bei dynamischer Bindung wird vor dem Zugriff auf den Wert von x im Rumpf von f die Bindung von x an 1 hergestellt; deshalb ergibt sich als Wert jetzt 3. \square

Statische Bindung hat zur Folge, dass sich Ausdrücke bei fester Bindung für ihre freien Variablen immer zu dem gleichen Ergebnis auswerten. Diese Eigenschaft nennt man auch *referentielle Transparenz*. Wie alle modernen funktionalen Programmiersprachen entscheiden wir uns bei FuL für statische Bindung.

Diese Festlegung hat Konsequenzen für die Implementierung der Funktionsanwendung. Betrachten wir die Auswertung der obigen Anwendung $h\ f\ 1$ und *call-by-need*-Parameterübergabe. Statische Bindung verlangt, dass alle freien Variablen in der rechten Seite **fun** $y \rightarrow x + y$ für f , hier die Variable x , wann immer diese rechte Seite ausgewertet wird, ihren Wert gemäß den Bindungen erhalten, die das textuell umgebende Konstrukt, hier das **let**, festlegt. Dort wird insbesondere x an den Wert 2 gebunden.

Eine solche Zuordnung wird auch *Umgebung* genannt. Damit bei jedem Antreffen einer *freien Variablen* eines Parameters die richtige Umgebung für diese Variable zur Verfügung steht, muss bei der Parameterübergabe mit dem Parameter e auch die zuständige Umgebung u übergeben werden. Das dafür zu bildende Paar (e, u) heißt *Abschluss* (Closure). Die Umgebung u in einem Abschluss (e, u) dient dazu, die freien Variablen von e richtig zu interpretieren. Auch bei *call-by-value* müssen gelegentlich Abschlüsse angelegt werden. Dies ist dann der Fall, wenn wie in Beispiel 3.2.3 Funktionen mit freien Variablen als Parameter übergeben werden.

Formal lässt sich die Menge $\text{free}(e)$ der freien Variablen induktiv über die Struktur von Programmausdrücken e so definieren:

$$\begin{aligned}
\text{free}(b) &= \emptyset && (b \text{ ein Basiswert}) \\
\text{free}(x) &= \{x\} && (x \text{ eine Variable}) \\
\text{free}(\square_1 e) &= \text{free}(e) \\
\text{free}(e_1 \square_2 e_2) &= \text{free}(e_1) \cup \text{free}(e_2) \\
\text{free}(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) &= \text{free}(e_0) \cup \text{free}(e_1) \cup \text{free}(e_2) \\
\text{free}(e' e_0 \dots e_{k-1}) &= \text{free}(e') \cup \text{free}(e_0) \cup \dots \cup \text{free}(e_{k-1}) \\
\text{free}(\text{fun } x_0 \dots x_{k-1} \rightarrow e') &= \text{free}(e') \setminus \{x_0, \dots, x_{k-1}\} \\
\text{free}(\text{let } x = e_1 \text{ in } e_0) &= \text{free}(e_1) \cup (\text{free}(e_0) \setminus \{x\}) \\
\text{free}(\text{let rec } x_1 = e_1 \text{ and } \dots \text{ and } x_n = e_n \text{ in } e_0) &= (\text{free}(e_0) \cup \dots \cup \text{free}(e_n)) \setminus \{x_1, \dots, x_n\}
\end{aligned}$$

Beispiel 3.2.4 Die Menge der freien Variablen des Ausdrucks:

$$\text{if } x \leq 1 \text{ then } 1 \text{ else } x \cdot \text{fac } (x - 1)$$

ist $\{x, \text{fac}\}$. Abstrahieren wir in diesem Ausdruck die Variable x , d.h. setzen wir: **fun** $x \rightarrow$ vor den Ausdruck, wird die Variable x *gebunden*. Als freie Variable bleibt dann nur noch fac übrig. \square

3.3 Die Architektur der MaMa

Im Folgenden skizzieren wir die Architektur einer virtuellen Maschine für FuL. Nach ihrem Erfinder Dieter Maurer heißt diese Maschine *MaMa* (**M**aurer **M**achine). Im Entwurf lehnen wir uns eng an die C-Maschine an, unsere virtuelle Maschine für imperative Sprachen. Wie die C-Maschine besitzt die MaMa einen Programmspeicher *C*, in dem MaMa-Programme abgelegt werden. Jede Zelle kann eine virtuelle Instruktion aufnehmen. Wieder enthält das Register *PC* (Program Counter, Befehlszähler) jeweils die Adresse des nächsten auszuführenden Befehls. Der Hauptausführungszyklus zur Abarbeitung von virtuellen Maschinenprogrammen ist der gleiche wie bei der C-Maschine:

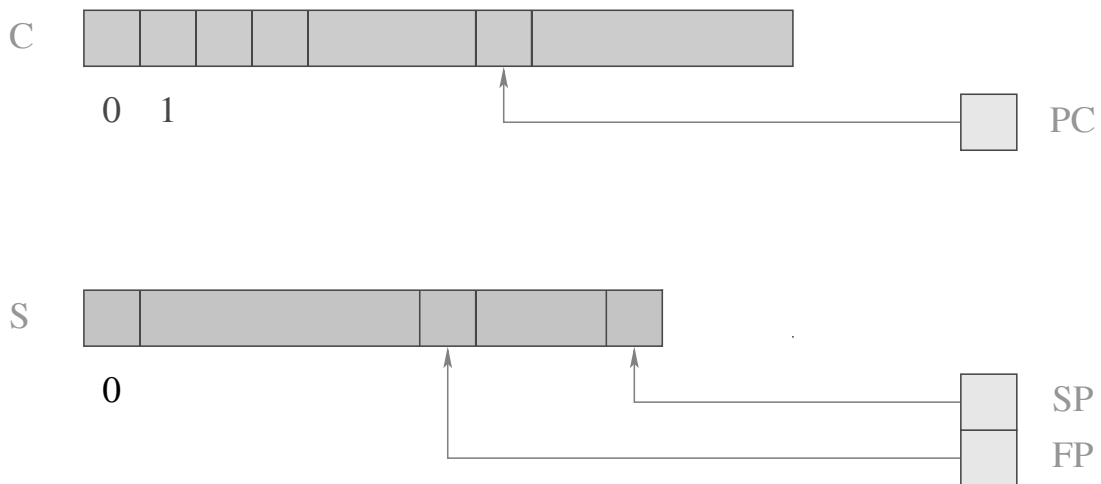


Abb. 3.1. Architektur der MaMa.

- Laden des nächsten Befehls;
- Erhöhung des *PC* um 1;
- Ausführen des geladenen Befehls.

Wie die C-Maschine besitzt die MaMa einen Befehl **halt**, der die Programmausführung beendet und die Kontrolle an das umgebende Betriebssystem zurückgibt.

Auch der Laufzeitkeller *S* mit den Registern *SP* und *FP* sind uns bereits bei der C-Maschine begegnet. Jede Zelle in *S* ist groß genug, um einen Basiswert oder auch eine Adresse aufzunehmen. Das Register *SP* zeigt jeweils auf die oberste belegte Zelle in *S*. Im Hinblick auf den gesamten für die Programmausführung zur Verfügung stehenden Platz sind wir bei der MaMa jedoch etwas großzügiger als bei der C-Maschine: wir nehmen an, dass für den Keller immer genügend Platz zur Verfügung steht.

Im Unterschied zur C-Maschine werden in der MaMa Werte grundsätzlich in der Halde *H* (Heap) angelegt (Abb. 3.2). Die Halde *H* kann man als *abstrakten Datentyp* aufzufassen, der insbesondere eine Operation:

$$\text{new } T(\text{args})$$

bereitstellt, welche ein neues MaMa-Objekt mit Etikett T und Argumenten $args$ in der Halde anlegt und einen Verweis darauf zurückliefert.

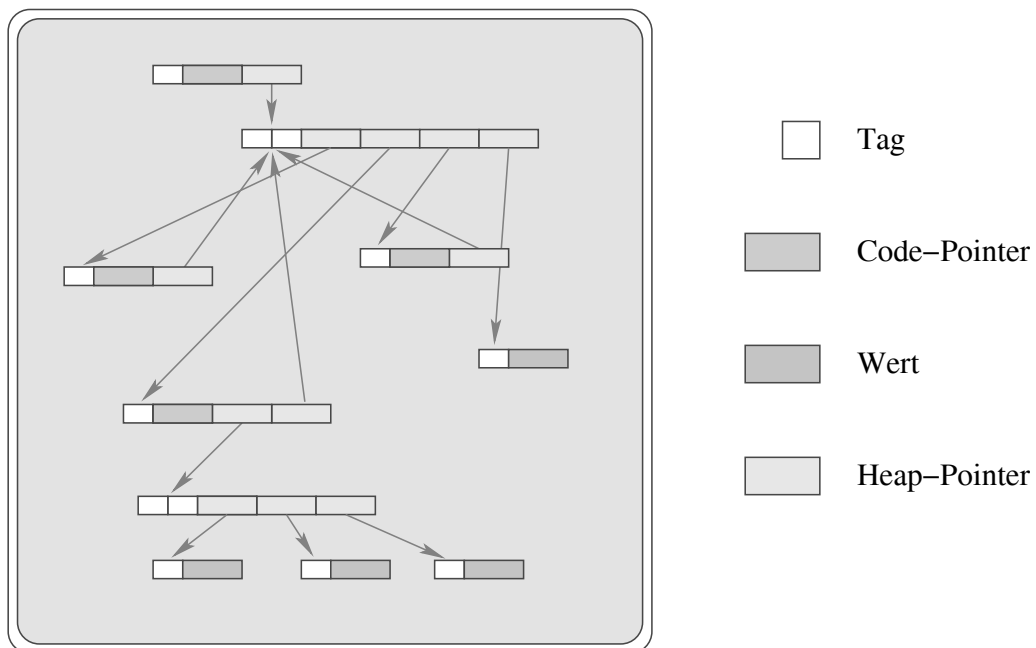


Abb. 3.2. Die Halde der MaMa.

Die für die Abspeicherung in der Halde benötigten Datenobjekte zeigt Abb. 3.3. Jedes solche Objekt besteht aus einem Etikett (engl. Tag) und eventuell mehreren Datenfeldern. B -Objekte benötigen wir zur Repräsentation von Basis-Werten. Sie bestehen aus dem Etikett „ B “, gefolgt von einem *int*-Wert. Verweise auf mehrere Werte können in einem V -Objekt zusammengefasst werden. Neben dem Etikett „ V “ und einer Angabe der Länge n enthält ein solches Objekt ein Feld v der angegebenen Länge. C - und F -Objekte dienen der Repräsentation von Abschlüssen und Funktionen. Neben ihrem entsprechenden Etikett enthalten sie jeweils einen Verweis cp (Code Pointer) auf einen zugehörigen Codeabschnitt sowie einen Verweis gp (Global Pointer) auf eine Repräsentation der Umgebung für die freien Variablen. Zusätzlich enthalten F -Objekte einen Verweis ap (Argument Pointer) auf einen Vektor bereits vorhandener Argumente für die Funktion.

Bevor wir uns daran machen, Übersetzungsschemata zur Erzeugung von MaMa-Code für Ausdrücke zu erstellen, müssen wir klären, was die zu erzeugende Folge von Instruktionen als Ergebnis liefern soll. Gemäß unserer Vereinbarung werden Werte in der Halde angelegt. Die Übersetzungsfunktion code_V für einen Ausdruck e soll deshalb dessen Wert ermitteln und einen Verweis darauf auf dem Keller zurückliefern.

Bei verzögerter Auswertung benötigen wir eine andere Instruktionsfolge. Diese soll ja gerade nicht zur Auswertung von e führen, sondern stattdessen einen Abschluss für e , d.h. ein entsprechendes C -Objekt für e in der Halde konstruieren und

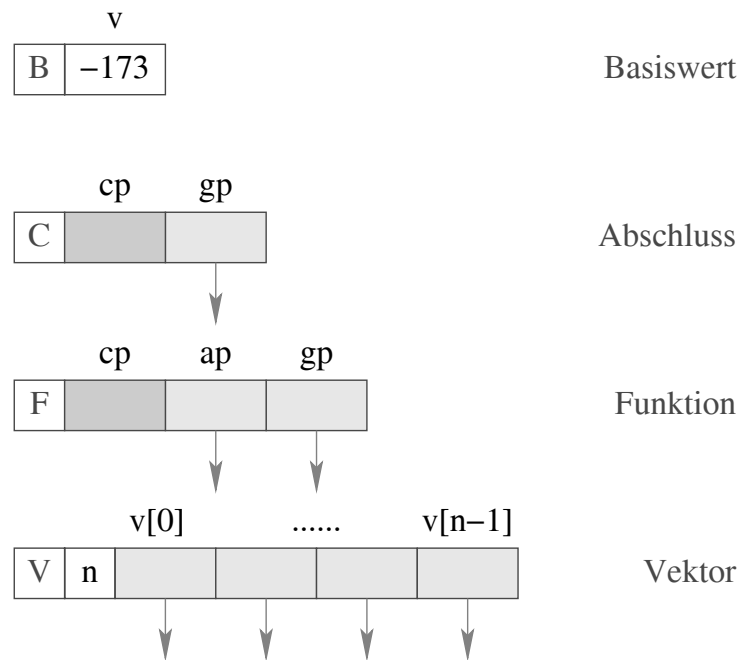


Abb. 3.3. Die Datenobjekte der MaMa.

einen Verweis darauf auf dem Keller zurückliefern. Dies leistet die Übersetzungsfunktion `codeC`, die wir in Abschnitt 3.11 näher vorstellen werden. Im Folgenden betrachten wir zuerst einmal die Übersetzungsschemata zur Definition von `codeV`.

3.4 Die Übersetzung einfacher Ausdrücke

Beginnen wir mit der Übersetzung *einfacher Ausdrücke*, das sind solche, die nur aus Basiswerten, Operatoren und Bedingungen aufgebaut sind. Ein Beispiel für einen solchen Ausdruck ist etwa:

$$\text{if } a \leq b + 1 \text{ then } b - a \text{ else } a - b$$

Als erstes bemerken wir, dass unsere Codeerzeugungsfunktionen wie bei der Übersetzung von `C` neben dem Programmstück, das übersetzt werden soll, als weiteres Argument eine Adressumgebung ρ benötigen, um auf die Bindungen der vorkommenden Variablen zugreifen zu können. Als zweites bemerken wir, dass die uns bereits bekannten arithmetischen Instruktionen wie etwa **add** oder **sub** ihre Argumente direkt oben auf dem Keller erwarten und nicht Verweise auf deren Werte. Das Gleiche gilt selbstverständlich auch für Vergleiche und bedingte Sprünge. Sofern sie ein Ergebnis haben, liefern alle diese Instruktionen ihr Ergebnis auch auf dem Keller zurück. Bei zusammengesetzten Ausdrücken erscheint es deshalb als äußerst ineffizient, die Werte der Argumente von Operatoren erst auf dem Keller zu haben, dann als `B`-Objekte in der Halde abzulegen, um sie anschließend sofort wieder auf den

Keller zu laden. Als Optimierung führen wir aus diesem Grund die Übersetzungsfunktion code_B ein, welche für einen Ausdruck e eine Instruktionsfolge liefert, die den Wert von e oben auf dem Keller hinterlässt. Diese Funktion ist ganz analog zur Übersetzungsfunktion zur Berechnung des R-Werts eines Ausdrucks in C:

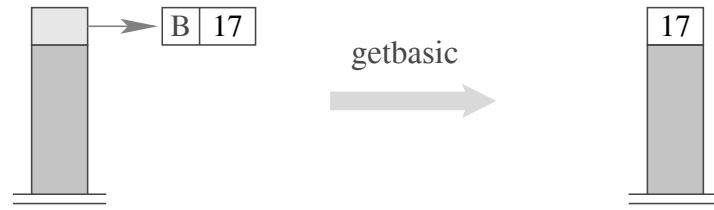
$$\begin{aligned}
 \text{code}_B b \rho kp &= \text{loadc } b \\
 \text{code}_B (\square_1 e) \rho kp &= \text{code}_B e \rho kp \\
 &\quad \text{op}_1 \\
 \text{code}_B (e_1 \square_2 e_2) \rho kp &= \text{code}_B e_1 \rho kp \\
 &\quad \text{code}_B e_2 \rho (kp + 1) \\
 &\quad \text{op}_2 \\
 \text{code}_B (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho kp &= \text{code}_B e_0 \rho kp \\
 &\quad \text{jumpz } A \\
 &\quad \text{code}_B e_1 \rho kp \\
 &\quad \text{jump } B \\
 &\quad A : \text{code}_B e_2 \rho kp \\
 &\quad B : \dots
 \end{aligned}$$

Dabei bezeichnen op_1 und op_2 die Instruktionen, die den jeweiligen Operator \square_1 bzw. \square_2 implementieren. Abweichend von der Übersetzungsfunktion code_R benötigt die Übersetzungsfunktion code_B neben der Adressumgebung ein weiteres Argument, den aktuellen *Kellerpegel* kp . Im Kellerpegel merken wir uns die Länge des lokalen Kellers. Diese Zusatzinformation werden wir später benötigen, um die lokalen Variablen adressieren zu können.

Für alle übrigen Ausdrücke berechnen wir erst den Wert im Heap und dereferenzieren dann:

$$\begin{aligned}
 \text{code}_B e \rho kp &= \text{code}_V e \rho kp \\
 &\quad \text{getbasic}
 \end{aligned}$$

Dabei liefert die Instruktion **getbasic** den Inhalt des B -Objekts, auf das der Verweis oben auf dem Keller zeigt (Abb. 3.4). Falls der Verweis nicht auf ein B -Objekt zeigt, wird ein Fehler gemeldet. Ein analoges Schema für die Übersetzung einfacher Ausdrücke in V -Code ist:

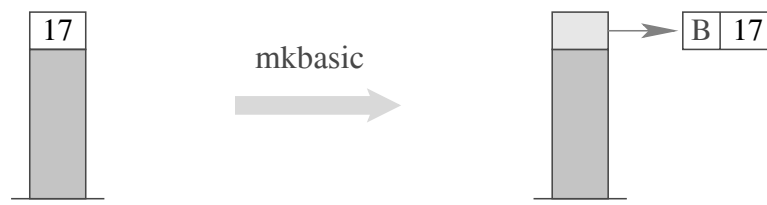


if ($H[S[SP]] \neq (B, _)$) error “not basic!”;
 else $S[SP] \leftarrow H[S[SP]].v$;

Abb. 3.4. Die Instruktion **getbasic**.

$\text{code}_V b \rho kp$	=	loadc b ; mkbasic
$\text{code}_V (\square_1 e) \rho kp$	=	$\text{code}_B e \rho kp$ op ₁ ; mkbasic
$\text{code}_V (e_1 \square_2 e_2) \rho kp$	=	$\text{code}_B e_1 \rho kp$ $\text{code}_B e_2 \rho (kp + 1)$ op ₂ ; mkbasic
$\text{code}_V (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho kp$	=	$\text{code}_B e_0 \rho kp$ jumpz A $\text{code}_V e_1 \rho kp$ jump B $A : \text{code}_V e_2 \rho kp$ $B : \dots$

Die Idee besteht darin, wann immer möglich, auf direkte Auswertung auf dem Keller umzuschalten, d.h. *B*-Code zu verwenden, und erst zum Schluss das Ergebnis der Berechnung in der Halde zu erzeugen. Dazu dient die Instruktion **mkbasic** (Abb. 3.5). Diese Instruktion erzeugt für den Wert, den sie oben auf dem Keller findet, ein neues *B*-Objekt.



$S[SP] \leftarrow \text{new } B(S[SP]);$

Abb. 3.5. Die Instruktion **mkbasic**.

Damit können wir bereits Ausdrücke übersetzen wie:

$$2 + (\text{if } 3 \text{ then } 4 - 5 \text{ else } 0)$$

Ausdrücke ohne Variablen sind jedoch nicht sehr spannend. Das Kernproblem, womit wir uns deshalb als nächstes beschäftigen müssen, ist der Zugriff auf Variablen.

3.5 Der Zugriff auf Variablen

Betrachten wir z.B. die Funktion f :

```
fun  $a \rightarrow$  let  $b = a \cdot a$ 
in  $b + c$ 
```

Die Funktion f benutzt die *globale* Variable c sowie die *lokalen* Variablen a (als formalem Parameter) und b (eingeführt durch **let**). Gemäß dem Konzept der statischen Bindung wird der Wert einer globalen Variablen bereits beim Anlegen der Funktion bestimmt und später nur nachgeschlagen.

Die Idee bei unserer Übersetzung besteht deshalb darin, die Bindungen der globalen Variablen in einem Vektor in der Halde zu verwalten, dem *Global-Vektor* (Global Vector). Beim Anlegen eines F -Objekts muss darum der Global-Vektor für die Funktion ermittelt und in der **gp**-Komponente abgelegt werden. Bei der Auswertung eines Ausdrucks zeigt das neue Register GP (der Global Pointer) auf den aktuellen Global-Vektor. Die lokalen Variablen verwalten wir dagegen auf dem Keller. Um zwischen lokalen und globalen Variablen unterscheiden zu können, haben deshalb die Adressumgebungen die Form:

$$\rho : \text{Vars} \rightarrow \{L, G\} \times \mathbb{Z}$$

Die globalen Variablen nummerieren wir geeignet durch und adressieren sie *relativ* zum Anfang des aktuellen Global-Vektors. Die Adressierung der lokalen Variablen dagegen hängt entscheidend von der Organisation unserer Kellerrahmen für Funktionen ab. Prinzipiell gibt es hier zwei Möglichkeiten.

Sei $e' e_0 \dots e_{m-1}$ die Anwendung einer Funktion e' auf Argumente e_0, \dots, e_{m-1} . Die erste Möglichkeit besteht darin, die aktuellen Parameter e_i von links nach rechts auszuwerten und in dieser Reihenfolge auf dem Keller zu hinterlassen (Abb. 3.6). Zeigt das Register FP auf die oberste Zelle vor Auswertung der Argumente, können die Argumente relativ zu FP adressiert werden. Stellt sich jedoch heraus, dass die Auswertung der Funktion e' eine Funktion f liefert, die bereits auf Argumente a_0, \dots, a_{k-1} angewandt ist, müssen diese neuen Parameter mühsam unten in den Keller eingefügt werden.

Alternativ könnten wir eine Strategie wählen, die bereits bei der Übersetzung von C zum Einsatz kam: wir werten die Argumente nicht von links nach rechts, sondern von rechts nach links aus (Abb. 3.7). Im Falle von C erwies sich diese Vorgehensweise als nützlich, weil eine C-Funktion mit weiteren optionalen Argumenten aufgerufen werden kann. Wertet sich bei dieser Anordnung der Argumente der

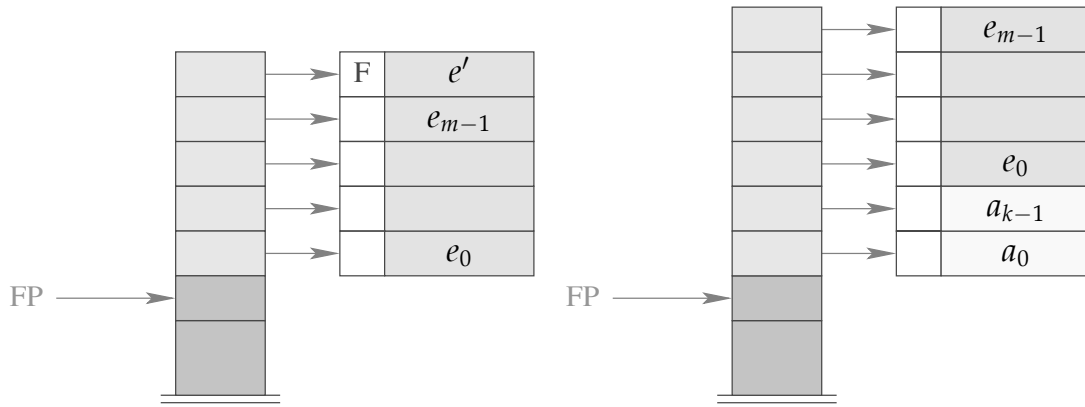


Abb. 3.6. Eine mögliche Kellerorganisation.

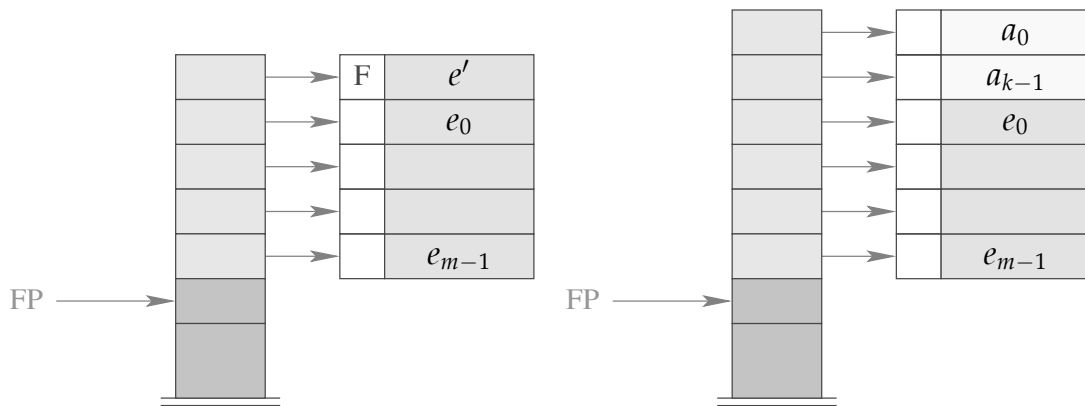


Abb. 3.7. Eine bessere Kellerorganisation.

FuL-Ausdruck e' zu einer Funktion f aus, die partiell auf Argumente a_0, \dots, a_{k-1} angewandt ist, können diese direkt oben auf den Keller gelegt werden.

Diese Anordnung hat nur einen kleinen Schönheitsfehler: eine Adressierung der formalen Parameter relativ zu einem festen Register ist möglich, falls wir ein weiteres Register einführen (s. Aufgabe 8). Dies werden wir hier jedoch nicht tun. Stattdessen wählen wir eine Adressierung relativ zum Kellerzeiger SP !

Das ist nicht so unsinnig, wie es auf den ersten Blick scheinen mag. Zwar ändert sich der SP während der Programmausführung ständig. Es zeigt sich jedoch, dass er an jedem Programmpunkt auf eine Adresse im Keller zeigt, die einen festen und bereits zur Übersetzungszeit bestimmbaren Abstand zu der obersten Kellerzelle SP_0 beim letzten Betreten einer Funktion besitzt (Abb. 3.8). Dieser feste Abstand $kp = SP - SP_0$ ist genau der *Kellerpegel*, den wir unseren Codeerzeugungsfunktionen als zusätzlichen Parameter mitgegeben haben.

Für die formalen Parameter x_0, x_1, x_2, \dots können wir nacheinander die nichtpositiven Relativadressen $0, -1, -2, \dots$, vergeben, d.h. $\rho x_i = (L, -i)$.

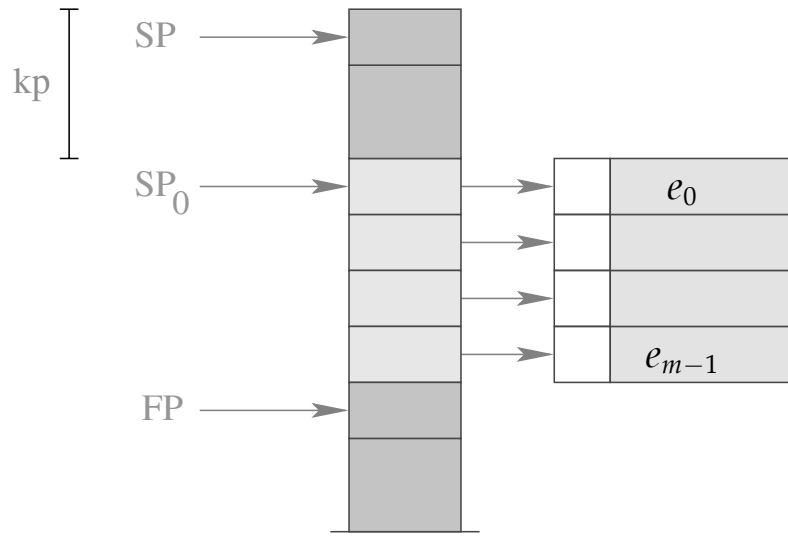


Abb. 3.8. Der Kellerpegel der MaMa.

Die *absolute* Adresse des i -ten formalen Parameters ergibt sich dann als

$$SP_0 + (-i) = (SP - kp) - i = SP - (kp + i)$$

Die lokalen, durch **let** oder **let rec** eingeführten Variablen y_1, y_2, y_3, \dots legen wir nacheinander oben auf dem Keller ab (Abb. 3.9). Die y_i erhalten darum positive

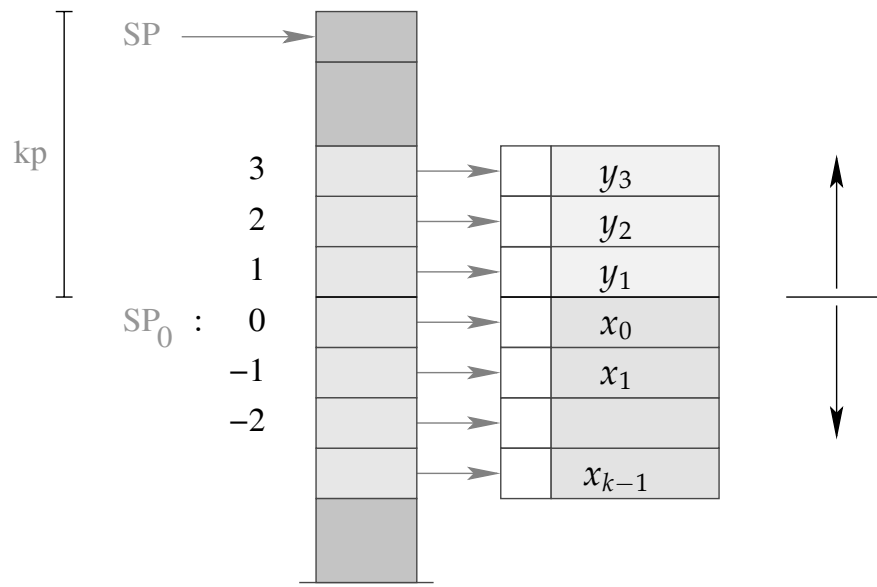


Abb. 3.9. Die lokalen Variablen der MaMa.

Relativadressen $1, 2, 3, \dots$, d.h.: $\rho y_j = (L, j)$. Die absolute Adresse von y_j ergibt sich dann als

$$SP_0 + j = (SP - kp) + j = SP - (kp - j)$$

Im Falle von CBN erzeugen wir damit für einen Variablenzugriff:

$$\text{code}_V x \rho kp = \text{getvar } x \rho kp \\ \text{eval}$$

Die Instruktion **eval** überprüft, ob der Wert bereits berechnet wurde oder ob seine Auswertung erst durchgeführt werden muss. Diese Instruktion behandeln wir später. Im Falle von CBV werden keinerlei Abschlüsse angelegt. Weil wir stets davon ausgehen können, dass die benötigten Werte bereits vorliegen, ist die Instruktion **eval** überflüssig.

Das Makro **getvar** selektiert in Abhängigkeit von der aktuellen Adressumgebung und dem Kellerpegel die tatsächliche Instruktion für den Variablenzugriff. Das Makro **getvar** ist definiert durch:

$$\text{getvar } x \rho kp = \text{match } \rho x \text{ with} \\ (L, i) \rightarrow \text{pushloc } (kp - i) \\ | (G, j) \rightarrow \text{pushglob } j$$

Zur Behandlung lokaler wie globaler Variablen benötigen wir neue Instruktionen. Für den Zugriff auf eine lokale Variable ist dies die Instruktion **pushloc** n (Abb. 3.10). Diese Instruktion lädt den Inhalt der Kellerzelle mit Adresse n relativ zum aktuellen SP oben auf den Keller. Seien sp und kp die aktuellen Werte des Keller-

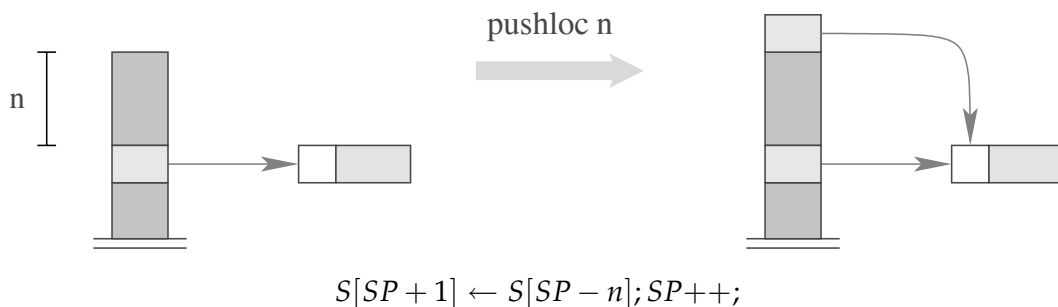
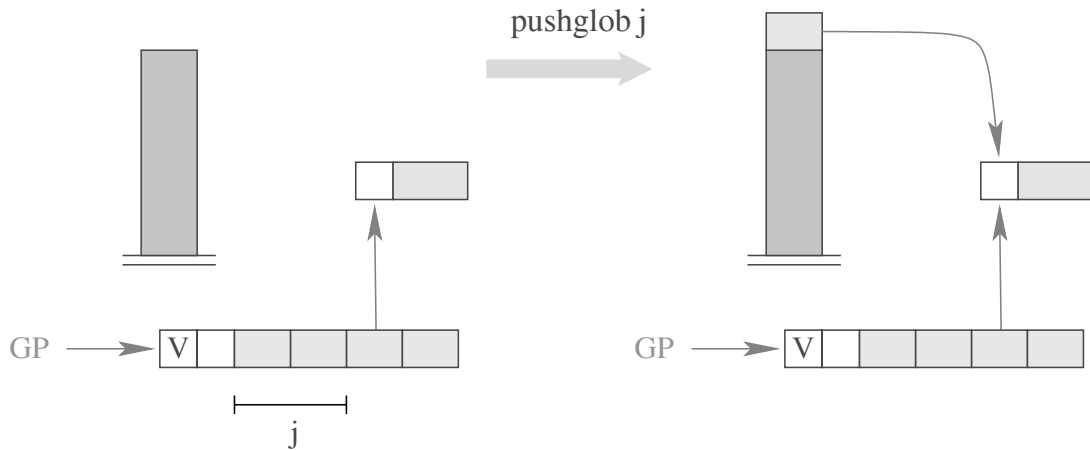


Abb. 3.10. Die Instruktion **pushloc** n .

zeigers SP bzw. Kellerpegels. Unser fester Referenzpunkt im Speicher ergibt sich dann als: $SP_0 = sp - kp$. Wir überzeugen uns, dass damit die Instruktion **pushloc** $(kp - i)$ tatsächlich den Inhalt der Speicherzelle mit Adresse $SP_0 + i$ lädt:

$$sp - (kp - i) = (sp - kp) + i = sp_0 + i$$

Der Zugriff auf die globalen Variablen ist viel einfacher. Hierzu führen wir die Instruktion **pushglob** j ein (Abb. 3.11). Diese Instruktion kopiert aus dem Vektor, auf den das Register GP zeigt, den Eintrag mit Index j auf den Keller.



$$S[SP + 1] \leftarrow H[GP].v[j]; SP++;$$

Abb. 3.11. Die Instruktion **pushglob j**.

Beispiel 3.5.1 Betrachten wir den Ausdruck $e \equiv (b + c)$ für eine Adressumgebung $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ und einen Kellerpegel $kp = 1$. Dann erhalten wir für CBN:

$\text{code}_V e \rho 1$	$= \text{getvar } b \rho 1$	$= 1$	pushloc 0
	eval	2	eval
	getbasic	2	getbasic
	$\text{getvar } c \rho 2$	2	pushglob 0
	eval	3	eval
	getbasic	3	getbasic
	add	3	add
	mkbasic	2	mkbasic

In der ersten Spalte haben wir alle Instruktionen bis auf diejenigen für die Variablenzugriffe bereits generiert. Die zweite Spalte enthält zusätzlich die entsprechend dem jeweiligen Kellerpegel zu wählenden Instruktionen für die Variablenzugriffe. Zur Verdeutlichung haben wir jeweils links neben die einzelnen Instruktionen den Kellerpegel vor der Ausführung der Instruktion vermerkt. Das Argument des Befehls **pushloc** ergibt sich dabei als:

$$kp - \rho b = 1 - 1 = 0$$

□

3.6 *let*-Ausdrücke

Zum Aufwärmen betrachten wir nun die Behandlung lokaler, *let*-definierter Variablen. Sei $e \equiv \text{let } x = e_1 \text{ in } e_0$ ein *let*-Ausdruck. Die Übersetzung von e muss

eine Befehlsfolge liefern, die die neue lokale Variable x auf dem Keller anlegt. Im Falle von CBV muss der Ausdruck e_1 ausgewertet und x an seinen Wert gebunden werden. Im Falle von CBN wird dagegen die Auswertung des Ausdrucks e_1 so lange verzögert, bis sein Wert tatsächlich benötigt wird. Deshalb wird nicht der Wert von e_1 berechnet, sondern ein Abschluss für e_1 hergestellt und ein Verweis darauf in x vermerkt. Schließlich muss der Ausdruck e_0 ausgewertet und sein Wert zurückgeliefert werden.

Entsprechend erzeugen wir im Fall von CBN:

$$\begin{aligned} \text{code}_V e \rho kp &= \text{code}_C e_1 \rho kp \\ &\quad \text{code}_V e_0 (\rho \oplus \{x \mapsto (L, kp + 1)\}) (kp + 1) \\ &\quad \text{slide } 1 \end{aligned}$$

Der *slide*-Befehl gibt am Ende den Platz für die lokale Variable x wieder frei. Im Falle von CBV muss der Wert der Variablen x dagegen sofort ermittelt werden. Darum benutzen wir in diesem Fall zur Codeerzeugung für den Ausdruck e_1 anstelle von code_C die Code-Funktion code_V .

Beispiel 3.6.1 Betrachten wir den Ausdruck:

$$e \equiv \text{let } a = 19 \text{ in let } b = a \cdot a \text{ in } a + b$$

für $\rho = \emptyset$ und $kp = 0$. Dann ergibt sich (für CBV):

0	loadc	19	3	getbasic	3	pushloc	1
1	mkbasic	3		mul	4	getbasic	
1	pushloc	0	2	mkbasic	4	add	
2	getbasic	2		pushloc	1	3	mkbasic
2	pushloc	1	3	getbasic	3	slide	1
					2	slide	1

□

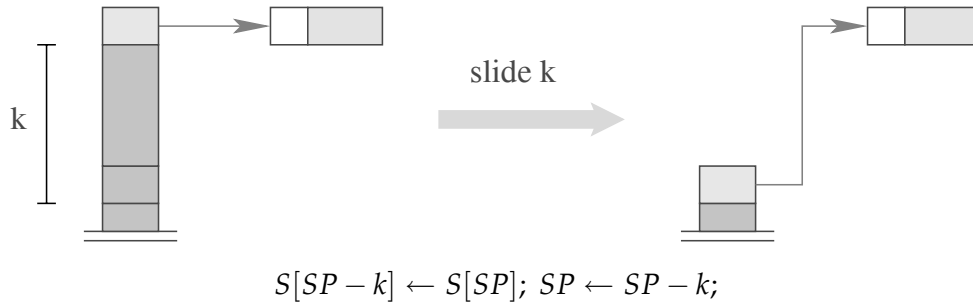
Der Befehl **slide** k gibt den Platz von k lokalen Variablen wieder frei (Abb. 3.12). Die allgemeinere Form dieser MaMa-Instruktion deutet schon an, dass es bei einer Folge ineinander geschachtelter *let*-Ausdrücke wie in Beispiel 3.6.1 ungeschickt ist, jede Variable einzeln freizugeben. Stattdessen lässt sich eine Folge von Instruktionen **slide** 1 der Länge k stets zu einem Befehl **slide** k zusammenfassen.

3.7 Funktionsdefinitionen

Wenden wir uns nun der Implementierung einer Funktion:

$$f \equiv \text{fun } x_0 \dots x_{k-1} \rightarrow e$$

zu. Für die Funktion f müssen wir Code erzeugen, die einen *funktionalen Wert* für f in der Halde anlegt. Das erfordert:

Abb. 3.12. Die Instruktion **slide k**.

- Erzeugen des Global-Vektors mit den Bindungen der in f freien Variablen;
- Erzeugen eines F -Objekts, das zusätzlich einen (zuerst leeren) Argumentvektor enthält sowie die Anfangsadresse des Codes zur Auswertung des Rumpfs; und schließlich
- Code zur Auswertung des Rumpfs.

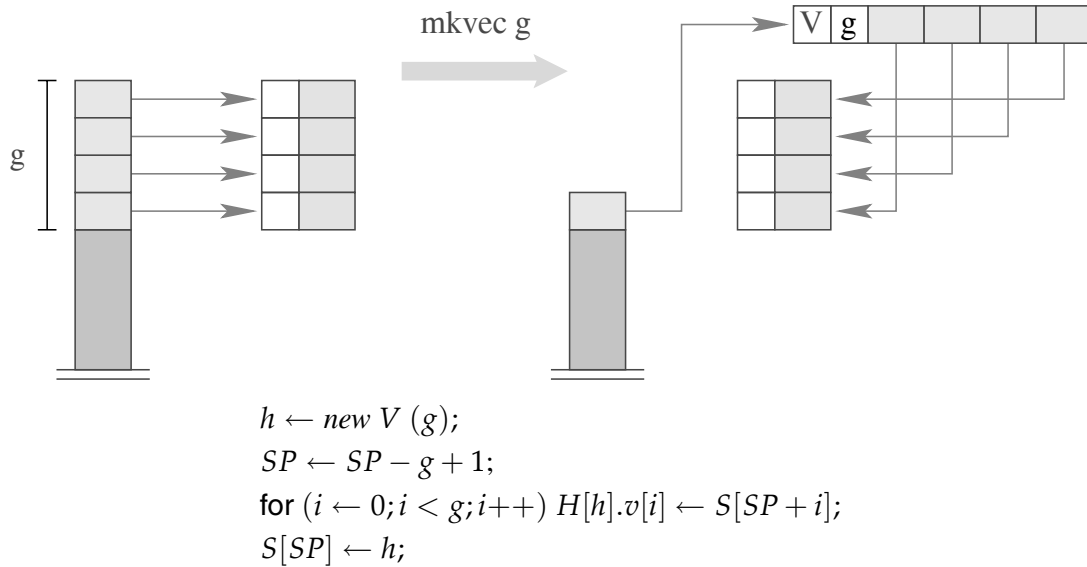
Diese Aufgaben können wir direkt in ein Codeerzeugungsschema umsetzen:

$$\begin{aligned} \text{code}_V(\text{fun } x_0 \dots x_{k-1} \rightarrow e) \rho kp = & \quad \text{getvar } z_0 \rho kp \\ & \text{getvar } z_1 \rho (kp + 1) \\ & \dots \\ & \text{getvar } z_{g-1} \rho (kp + g - 1) \\ & \text{mkvec } g \\ & \text{mkfunval } A \\ & \text{jump } B \\ A : & \text{targ } k \\ & \text{code}_V e \rho' 0 \\ & \text{return } k \\ B : & \dots \end{aligned}$$

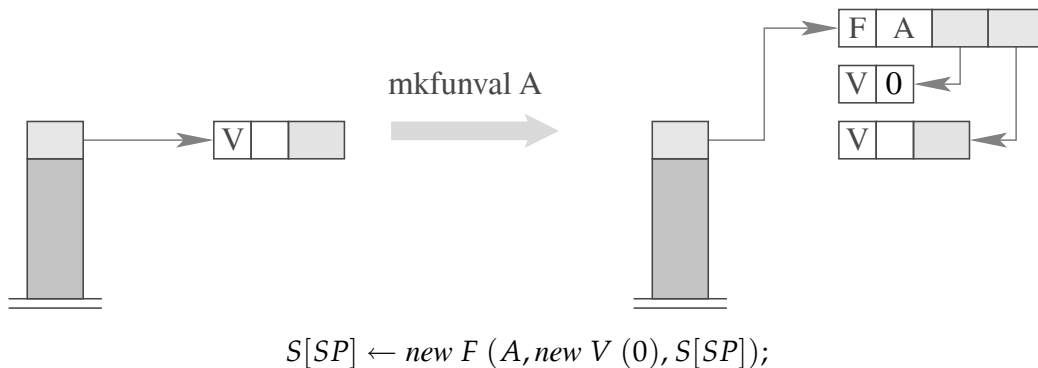
wobei

$$\begin{aligned} \{z_0, \dots, z_{g-1}\} &= \text{free}(\text{fun } x_0 \dots x_{k-1} \rightarrow e) \quad \text{und} \\ \rho' &= \{x_i \mapsto (L, -i) \mid i = 0, \dots, k-1\} \cup \\ &\quad \{z_j \mapsto (G, j) \mid j = 0, \dots, g-1\} \end{aligned}$$

Beachten Sie, dass die Menge der globalen Variablen statisch bekannt ist: sie kann mithilfe unserer Funktion **free** bestimmt werden. Die Folge der Aufrufe des Makro **getvar** legt die Verweise auf die globalen Variablen der Funktion nacheinander auf den Keller. Für jeden weiteren Verweis muss der Kellerpegel entsprechend erhöht werden. Der Befehl **mkvec** g (Abb. 3.13) stellt aus der Abfolge der Verweise auf die globalen Variablen oben auf dem Keller einen Vektor in der Halde her. Er entfernt die entsprechenden Zellen vom Keller und hinterlegt dort einen Verweis auf das neu erzeugte V -Objekt. Die Instruktion **mkfunval** A (Abb. 3.14) konstruiert aus der An-

Abb. 3.13. Die Instruktion **mkvec g**.

fangsadresse A für die Auswertung der Funktion und dem Vektor für die Bindungen der globalen Variablen das F -Objekt für die Funktion. Zusätzlich initialisiert diese Instruktion noch die Argumentvektor-Komponente des neuen F -Objekts mit einem leeren Vektor.

Abb. 3.14. Die Instruktion **mkfunval A**.

Unser Übersetzungsschema stellt auch den Code bereit, der bei einem Funktionsaufruf ausgeführt werden soll. Das ist sinnvoll, weil dieser Code dieselbe Zuordnung von globalen Variablen zu Positionen im Global-Vektor verwenden muss, die beim Einpacken der Bindungen benutzt wurde. Andererseits darf diese Instruktionsfolge bei Erstellung des F -Objekts selbst keineswegs ausgeführt werden: deshalb wird dieser Codeabschnitt übersprungen, um dahinter an der Stelle B fortzufahren.

Der Code für die Abarbeitung eines Aufrufs der Funktion steht somit zwischen den Marken A und B . Im Wesentlichen besteht er aus dem V -Code für den Rumpf,

mit einer Instruktion **targ** k davor und einer Instruktion **return** k dahinter, wobei k die Anzahl der Argumente ist, die für eine Abarbeitung des Rumpfs erforderlich sind. Diese beiden Instruktionen behandeln wir im Detail in Kapitel 3.9. Hier sei nur so viel verraten, dass die Instruktion **targ** k zur Behandlung von Unterversorgung mit Argumenten und die Instruktion **return** k für die Endbehandlung des Aufrufs und eventuelle Überversorgung mit Argumenten zuständig ist.

Beispiel 3.7.1 Betrachten wir die Funktion $f \equiv \text{fun } b \rightarrow a + b$ in einer Adressumgebung $\rho = \{a \mapsto (L, 1)\}$ mit Kellerpegel $kp = 1$. Dann liefert der Aufruf $\text{code}_V f \rho 1$:

1	pushloc 0	0	pushglob 0	2	getbasic
2	mkvec 1	1	eval	2	add
2	mkfunval A	1	getbasic	1	mkbasic
2	jump B	1	pushloc 1	1	return 1
0	A : targ 1	2	eval	2	B : ...

□

3.8 Funktionsanwendungen

Funktionsanwendungen entsprechen Funktionsaufrufen in C. Die für eine Funktionsanwendung erzeugte Befehlsfolge muss dafür sorgen, dass bei Eintritt in die Funktion auf dem Keller die Verhältnisse herrschen, die bei der Übersetzung der Funktionsdefinition vorausgesetzt werden. Notwendige Aktionen zur Auswertung einer Funktionsanwendung $e' e_0 \dots e_{m-1}$ sind damit:

- Anlegen eines Kellerrahmens;
- Parameterübergabe, d.h. für CBV Auswerten der aktuellen Parameter bzw. im Fall von CBN Anlegen von Abschlüssen für die aktuellen Parameter;
- Auswerten der Funktion e' zu einem F -Objekt;
- Anwenden der Funktion.

Für CBN gibt das Anlass zu dem folgenden Übersetzungsschema:

$$\begin{aligned} \text{code}_V (e' e_0 \dots e_{m-1}) \rho kp = & \text{mark } A \\ & \text{code}_C e_{m-1} \rho (kp + 3) \\ & \text{code}_C e_{m-2} \rho (kp + 4) \\ & \dots \\ & \text{code}_C e_0 \rho (kp + m + 2) \\ & \text{code}_V e' \rho (kp + m + 3) \\ & \text{apply} \\ & A : \dots \end{aligned}$$

Tatsächlich ist dieses Schema ganz analog dem Übersetzungsschema für Aufrufe von C-Funktionen aufgebaut. Der Befehl **mark** A legt einen Kellerrahmen für den Aufruf an. Dann werden die aktuellen Parameter von rechts nach links abgearbeitet. Im Falle einer CBN-Semantik erzeugen wir Code zum Anlegen der entsprechenden Abschlüsse, deshalb die Benutzung von `codeC` für die e_i . Im Falle einer CBV-Semantik müssen wir stattdessen Code zum Auswerten der Argumente erzeugen, d.h. statt der `codeC`-Funktion die `codeV`-Funktion für die e_i verwenden. Beachten Sie, dass sich mit jeder Referenz, die auf dem Keller abgelegt wird, der Kellerpegel für den nächsten Aufruf einer Codeerzeugungsfunktion erhöht.

Nachdem die aktuellen Parameter übersetzt sind, muss Code zur Ermittlung der aufzurufenden Funktion erzeugt werden. Dies besorgt der Aufruf der Codeerzeugungsfunktion `codeV` für e' . Zur Laufzeit wird diese Instruktionsfolge auf dem Keller einen Verweis auf ein F -Objekt hinterlassen. Wir benötigen darum einen weiteren Befehl, der dieses F -Objekt auspackt und den entsprechenden Code zur Ausführung der Funktion anspringt. Das leistet die Instruktion **apply**.

Beispiel 3.8.1 Für einen Aufruf (f 42) in der Adressumgebung $\rho = \{f \mapsto (L, 2)\}$ bei Kellerpegel $kp = 2$ liefert unser Schema für CBV:

2 mark A	6 mkbasic	7 apply
5 loadc 42	6 pushloc 4	3 $A : \dots$

Als etwas größeres zusammenhängendes Beispiel betrachten wir den Ausdruck:

let $a = 17$ **in** **let** $f = \text{fun } b \rightarrow a + b$ **in** f 42

Teile dieses Ausdrucks haben wir bereits in den vorigen Beispielen behandelt. Bei CBV-Semantik der Programmiersprache erhalten wir für $kp = 0$ insgesamt:

0 loadc 17	2 jump B	2 getbasic	5 loadc 42
1 mkbasic	0 $A : \text{targ } 1$	2 add	5 mkbasic
1 pushloc 0	0 pushglob 0	1 mkbasic	6 pushloc 4
2 mkvec 1	1 getbasic	1 return 1	7 apply
2 mkfunval A	1 pushloc 1	2 $B : \text{mark } C$	3 $C : \text{slide } 2$

□

Bevor wir an die Implementierung der neuen Instruktionen gehen können, müssen wir uns über die Organisation eines Kellerrahmens zur Auswertung eines Funktionsaufrufs im Klaren sein (Abb. 3.15). Wie die C-Maschine benötigt auch die MaMa drei organisatorische Zellen, in denen die Werte der Register GP , FP und PC vor dem Aufruf gerettet werden, um sie nach Beendigung des Aufrufs restaurieren zu können. Wir verständigen uns darauf, dass der Rahmenzeiger FP auf die oberste organisatorische Zelle zeigt. Die Argumente der Funktion liegen damit oberhalb des FP . Oberhalb der Argumente beginnt der lokale Keller für Zwischenergebnisse und weitere lokale Variablen.

Damit können wir nun bestimmen, was die Instruktion **mark** A tun soll (Abb. 3.16). Im Gegensatz zu dem entsprechenden Befehl der C-Maschine rettet er hier

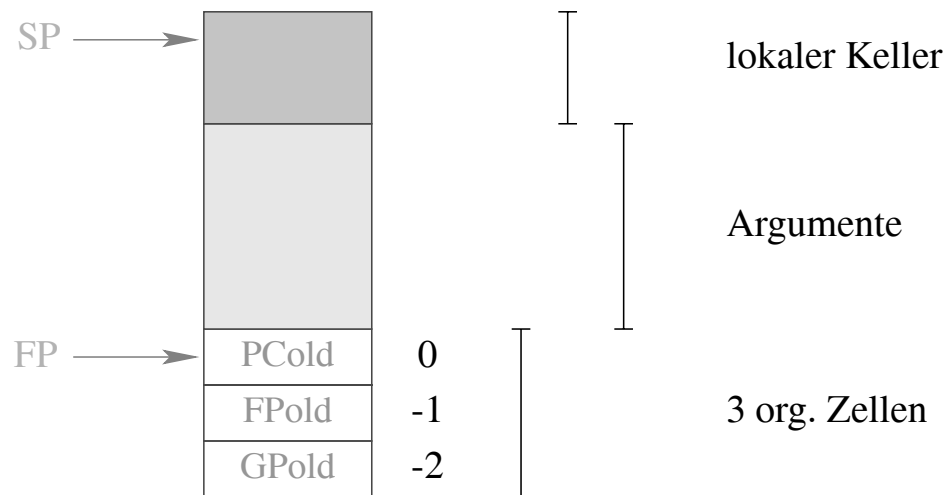
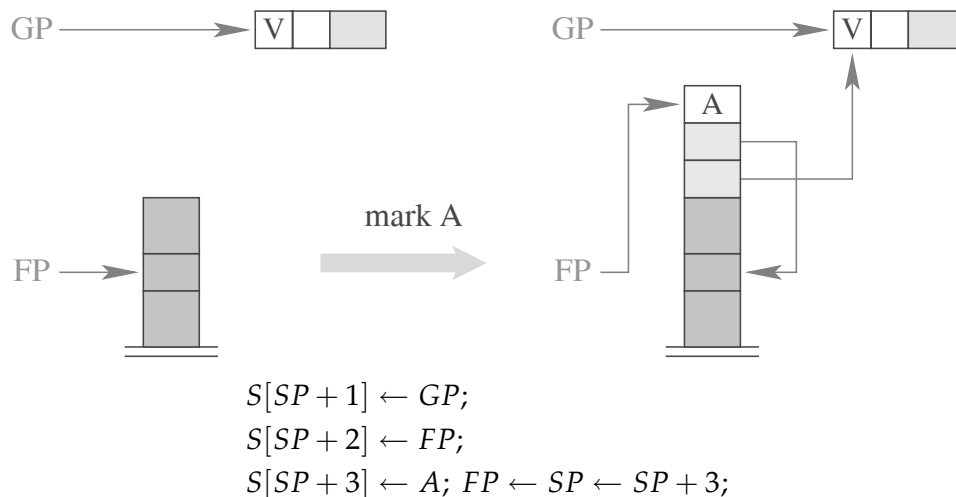
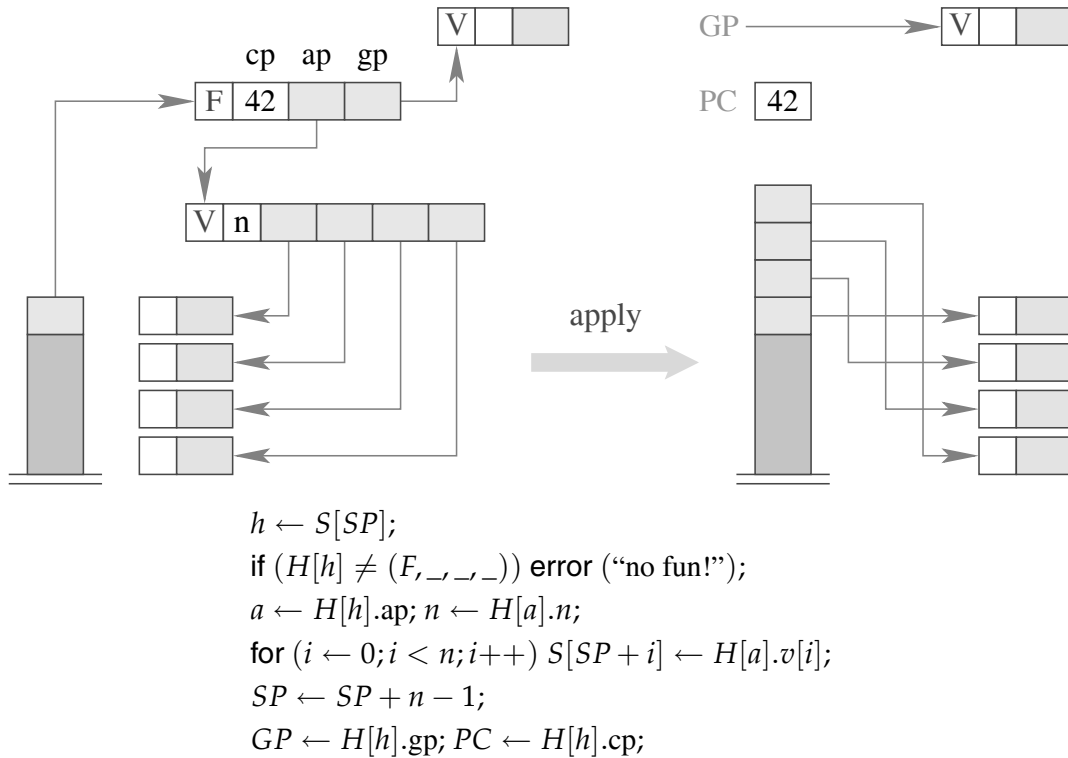


Abb. 3.15. Der Kellerrahmen der MaMa.

Abb. 3.16. Die Instruktion **mark A**.

alle drei Register. Wie bei der C-Maschine legen wir die Rücksprungadresse als letzte auf den Keller. Deshalb benötigt er als Argument die Adresse, an der die Programmausführung nach Rückkehr aus dem Funktionsaufruf fortfahren soll. Weiterhin setzt er bereits den neuen Rahmenzeiger auf die oberste organisatorische Zelle, hier den geretteten *PC*.

Der Befehl **apply** muss das *F*-Objekt, auf das (hoffentlich) oben auf dem Keller ein Verweis liegt, auspacken und an der dort angegebenen Adresse fortfahren (Abb. 3.17). Beachten Sie, dass das 0-te Element des Argumentvektors zuerst auf den Keller gelegt wird. Diese Konvention müssen wir berücksichtigen, um eine unterversorgte Funktion zu einem *F*-Objekt einzupacken. Dieses 0-te Element muss also die *äußerste* Argumentreferenz der unterversorgten Funktion darstellen.

Abb. 3.17. Die Instruktion **apply**.

Von der Gesamtaufgabe, Funktionen zu implementieren, haben wir jetzt die folgenden Teilaufgaben behandelt: die Erzeugung von *F*-Objekten aus Funktionsdefinitionen und darin geschachtelt die Übersetzung des Rumpfes mit der richtigen Umgebung; das Anlegen eines Kellerrahmens für die Funktionsanwendung und die anschließende Ablage von Verweisen auf die Argumente, und endlich den Anspruch des Codes für die Funktion. Übrig bleibt noch der „Rahmen“ um die Übersetzung des Funktionsrumpfes, nämlich die Befehle **targ** *k* und **return** *k*. Diese behandeln wir im nächsten Abschnitt.

3.9 Unter- und Überversorgung mit Argumenten

Der erste Befehl, der zur Laufzeit nach einem **apply** ausgeführt wird, ist **targ** *k*. Er überprüft, ob bereits genügend Argumente vorliegen, um den Funktionsrumpf auszuführen. Die Anzahl der Argumente ist gegeben durch die Differenz $SP - FP$. Sind genügend Argumente vorhanden, d.h. ist $SP - FP \geq k$, soll der Rumpf normal betreten werden. Sind nicht genügend Argumente vorhanden, d.h. ist $SP - FP < k$, liefert **targ** *k* als Ergebnis ein neues *F*-Objekt zurück.

Der Befehl **targ** *k* ist ein relativ komplizierter Befehl. Um seinen Effekt besser zu verstehen, zerlegen wir seine Ausführung darum in mehrere Schritte:

```

targ  $k = \text{if } (SP - FP < k) \{$ 
    mkvec0;           // Anlegen des Argument-Vektors
    wrap;             // Anlegen des  $F$ -Objekts
    popenv;          // Aufgeben des Kellerrahmens
 $\}$ 

```

Die Zusammenfassung dieser festen Schrittabfolge zu einem Befehl können wir wieder als eine Optimierung auffassen.

Die erste Teilaufgabe, die wir im Falle von Unterversorgung zu erledigen haben,

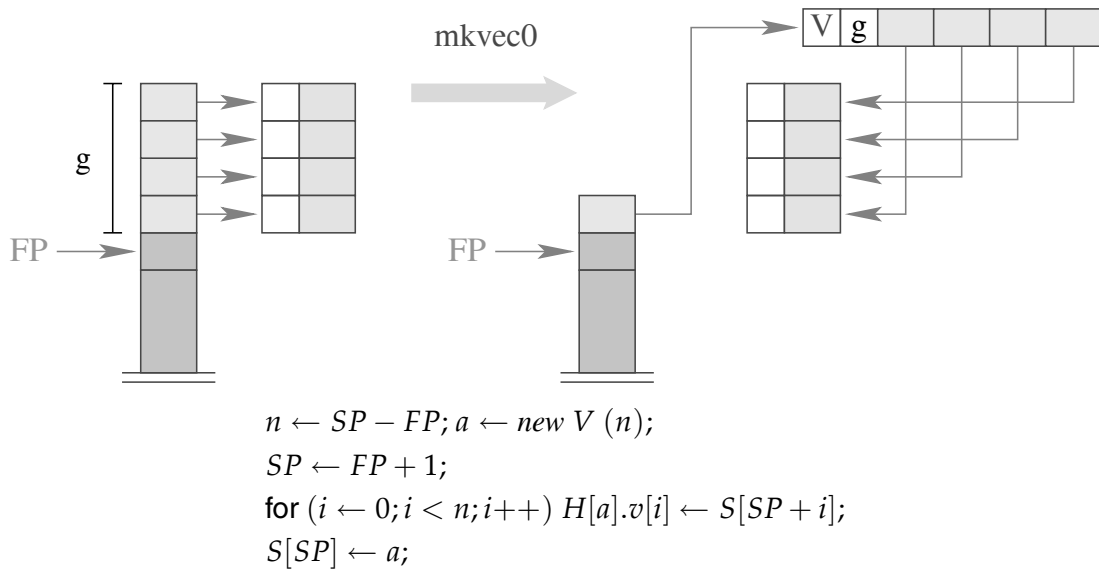
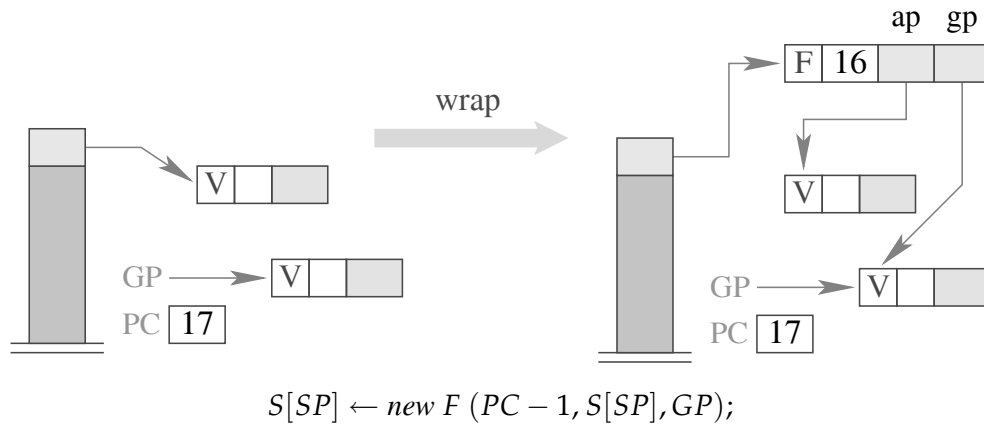
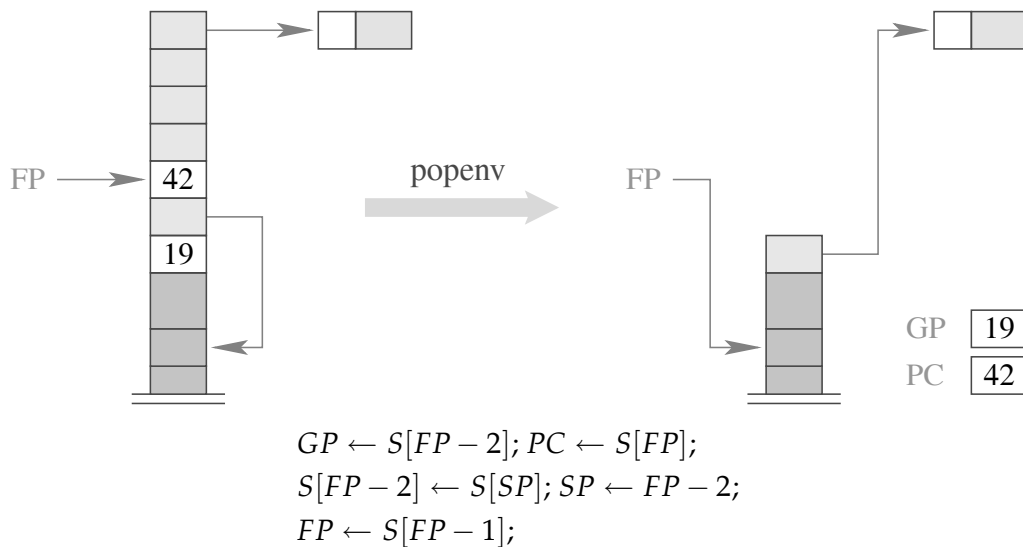


Abb. 3.18. Die Instruktion **mkvec0**.

ist das Einpacken der Argumentverweise zwischen SP und FP in einen Vektor. Das erledigt unsere Mikroinstruktion **mkvec0** (Abb. 3.18).

Anschließend wird dieser Argumentvektor zusammen mit dem aktuellen GP wieder zu einem F -Objekt eingepackt. Dazu haben wir die Mikroinstruktion **wrap** eingeführt (Abb. 3.19). Beachten Sie, dass diese Instruktion die Anfangsadresse des Codes zur Ausführung des Funktionsaufrufs mit einpacken muss, d.h. in unserem Fall die Adresse der Instruktion **targ** k . Da sich während der Abfolge der Mikroinstruktionen der PC nicht ändert, ist dies gerade der aktuelle Inhalt des Registers PC , vermindert um 1.

Bleibt schließlich noch, den Kellerrahmen aufzugeben und das konstuierte F -Objekt als Ergebnis zurückzuliefern. Das leistet die Mikroinstruktion **popenv** (Abb. 3.20). Beachten Sie, dass unsere Mikroinstruktion **popenv** geringfügig allgemeiner definiert ist als für ihre Anwendung in **targ** k erforderlich: zwischen dem Rückgabewert oben auf dem Keller und den organisatorischen Zellen dürfen beliebig viele weitere Zellen liegen. Bei Unterversorgung mit Argumenten erhalten wir damit für **targ** k die Transformation aus Abb. 3.21.

Abb. 3.19. Die Instruktion **wrap**.Abb. 3.20. Die Instruktion **popenv**.

Die Instruktion **return** k sorgt für die Endbehandlung von Funktionsanwendungen. Ihr Argument k gibt die Anzahl der durch die abgeschlossene Bearbeitung konsumierten Argumente an. Sie behandelt zwei Fälle. Stehen im Kellerrahmen genau so viele Verweise auf Argumente, wie die angewandte Funktion benötigte, kann der aktuelle Kellerrahmen freigegeben und das Ergebnis zurückgeliefert werden. Stehen dagegen mehr Verweise auf Argumente im Kellerrahmen, als die Funktion benötigte, liegt eine *Übersorgung* der Funktion mit Argumenten vor. In diesem Fall sollte der aktuelle Aufruf mit den bereits konsumierten Argumenten eine Funktion als Ergebnis geliefert haben, welche in der Lage ist, weitere Argumente zu konsumieren. Da oberhalb der Argumente das Ergebnis der bisherigen Funktionsauswertung auf dem Keller liegt, ergibt sich die Anzahl der vorhandenen Argumente zu $SP - FP - 1$. Auch den relativ komplexen Befehl **return** k definieren wir mithilfe überschaubarer Mikroinstruktionen:

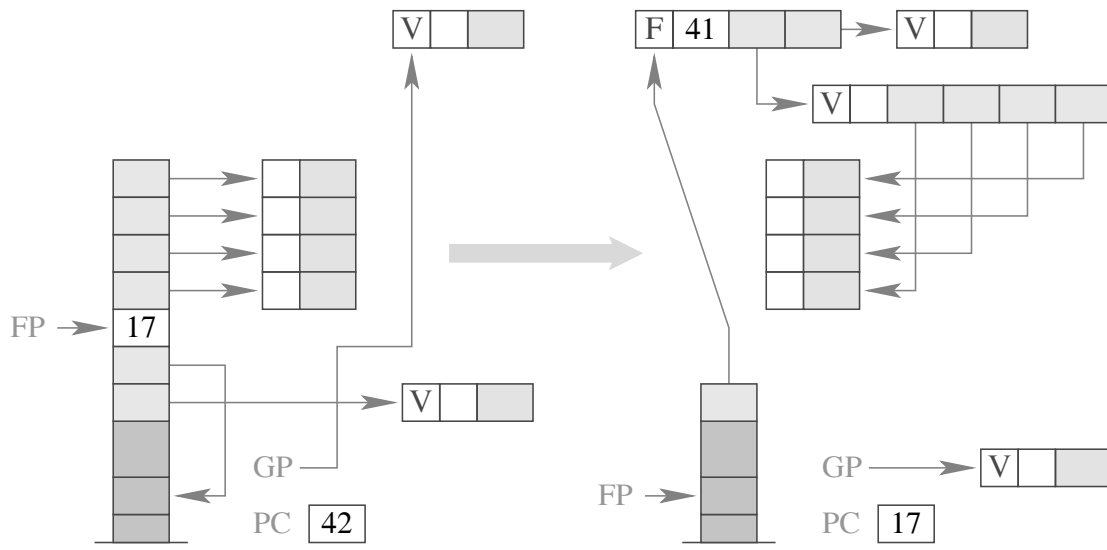


Abb. 3.21. Die Instruktion **targ k** bei Unterversorgung mit Argumenten.

```

return  $k = \text{if } (SP - FP - 1 \leq k)$ 
    popenv;                                // Aufgeben des Kellerrahmens
    else {                                  // Es gibt noch weitere Argumente
        slide  $k$ ;
        apply;                            // erneuter Aufruf
    }

```

Tatsächlich verwenden wir hier ausschließlich bereits definierte (Mikro-) Instruktionen. Das Verhalten von **return k** für den Fall der richtigen Anzahl von Argumenten zeigt Abb. 3.22. In diesem Fall erledigt die Arbeit die Mikroinstruktion **popenv**.

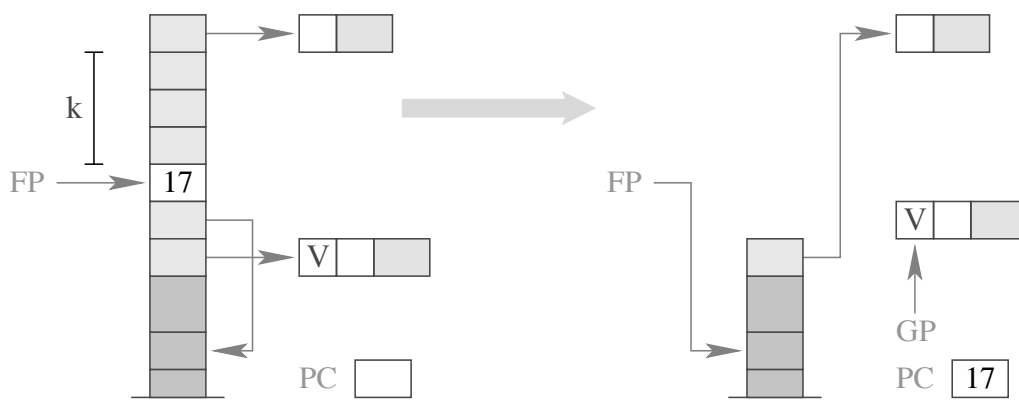


Abb. 3.22. Die Instruktion **return k** ohne Überversorgung mit Argumenten.

In dem komplizierteren Fall, dass bei Erreichen der Instruktion **return** k weitere, unverbrauchte Argumente auf dem Keller liegen, entfernt die Instruktion **slide** k die obersten k nun nicht mehr benötigten Argumentverweise. Die Auswertung des oben auf dem Keller liegenden F -Objekts wird schließlich durch die Instruktion **apply** veranlasst. Den Gesamteffekt für diesen Fall zeigt Abb. 3.23. Bemerken Sie, dass

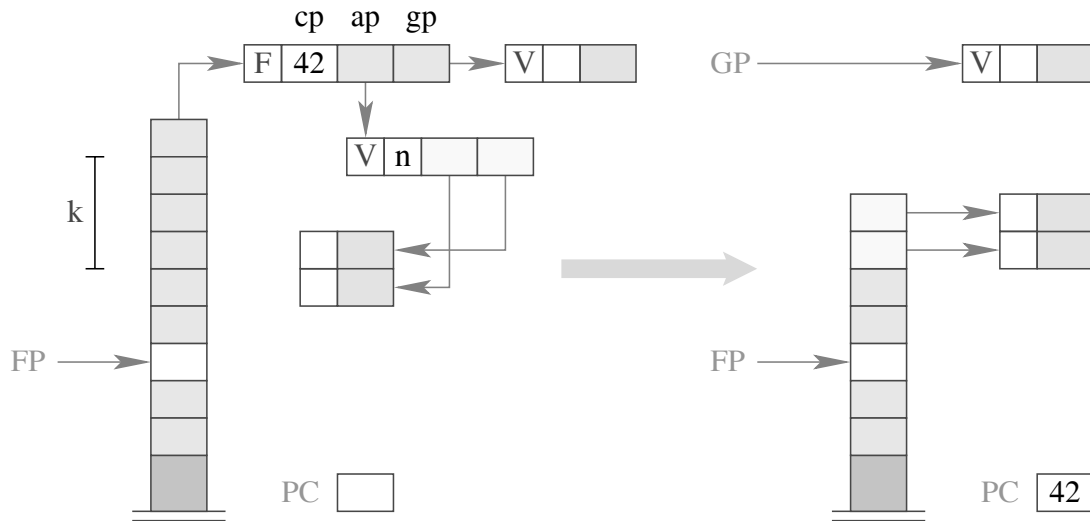


Abb. 3.23. Die Instruktion **return** k bei Überversorgung mit Argumenten.

dieser Folgeaufruf im selben Kellerrahmen erfolgt wie sein Vorgänger.

3.10 Rekursive Variablendefinitionen

Für eine rekursive Variablendefinition

$$e \equiv \text{let rec } y_1 = e_1 \text{ and } \dots \text{ and } y_n = e_n \text{ in } e_0$$

muss die Übersetzung eine Befehlsfolge liefern, die folgendes leistet:

- Zuerst müssen die lokalen Variablen y_1, \dots, y_n angelegt werden.
- Dann müssen die Ausdrücke e_1, \dots, e_n ausgewertet (im Fall von CBV) bzw. Abschlüsse für diese Ausdrücke hergestellt und die Variablen y_i entsprechend gebunden werden.
- Schließlich muss der Ausdruck e_0 ausgewertet und sein Wert zurückgeliefert werden.

Dabei müssen wir darauf achten, dass in einem *letrec*-Ausdruck die Variablen *simultan* definiert werden. Das bedeutet z.B., dass bereits im Ausdruck e_1 für die erste Variable y_1 sämtliche Variablen y_1, \dots, y_n vorkommen dürfen. Das wird dadurch ermöglicht, dass vor der eigentlichen Definition der einzelnen Variablen für jede Variable y_i bereits eine Referenz auf einen leeren Abschluss, genannt *Dummy-Abschluss*,

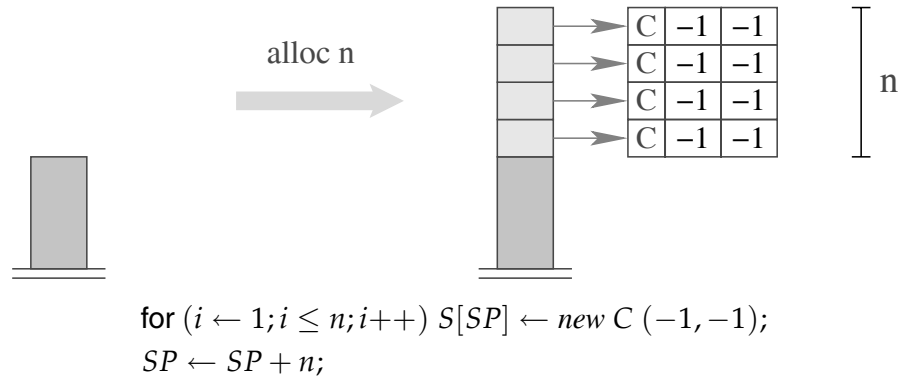


Abb. 3.24. Die Instruktion **alloc** n .

auf den Keller gelegt wird. Das leistet die MaMa-Instruktion **alloc** n (Abb. 2.29). Die leeren Abschlüsse werden später mit den richtigen Werten (bzw. bei CBN den richtigen Abschlüssen) überschrieben. Das leistet die Instruktion **rewrite** j (Abb. 3.25). Das Argument j der Instruktion ist die Differenz $a_2 - a_1$ zwischen zwei Adressen

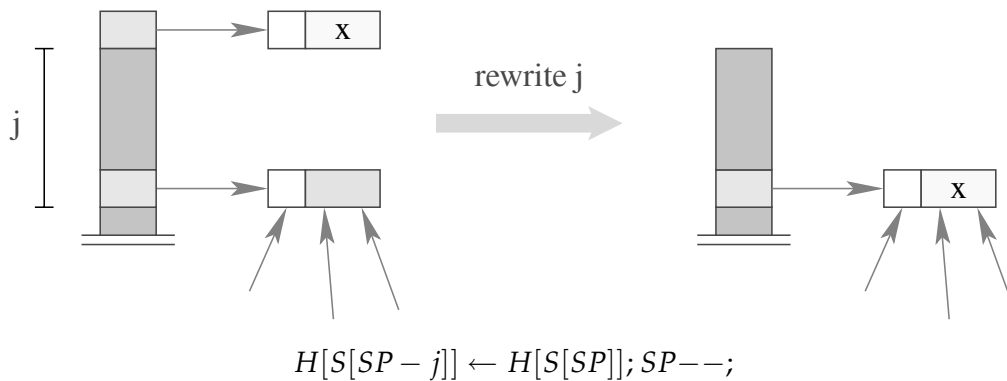


Abb. 3.25. Die Instruktion **rewrite** j .

a_1, a_2 im Keller. $S[a_1]$ enthält die Referenz auf einen Dummy-Abschluss, der mit dem Haldenobjekt überschrieben werden soll, dessen Referenz in $S[a_2]$ oben auf dem Keller steht. Damit wird also der Wert $H[S[a_1]]$ überschrieben; die Referenz $S[a_1]$ dagegen bleibt erhalten!

Insgesamt erhalten wir für rekursive Variablendefinitionen e das Schema:


```

codeV e ρ kp = alloc n      // legt die lokalen Variablen an
               codeC e1 ρ' (kp + n)
               rewrite n
               ...
               codeC en ρ' (kp + n)
               rewrite 1
               codeV e0 ρ' (kp + n)
               slide n      // gibt die lokalen Variablen auf

```

wobei $\rho' = \rho \oplus \{y_i \mapsto (L, kp + i) \mid i = 1, \dots, n\}$. Im Falle von CBV benutzen wir für die Ausdrücke e_1, \dots, e_n ebenfalls die Codeerzeugungsfunktion `codeV`. Sie sollten sich allerdings klar machen, dass im Falle von CBV die Auswertung der Ausdrücke e_j keinesfalls auf den Wert einer der Variablen y_i mit $i \geq j$ zugreifen darf! Dies ist gewährleistet, wenn wir bei CBV nur rekursive Definitionen von *Funktionen* zulassen.

Die leeren Abschlüsse werden der Reihe nach überschrieben. Bei Beginn der Ausführung des Codes für die e_i wie für den Hauptausdruck e_0 ist der Kellerpegel stets um n erhöht. Der Verweis auf den ersten leeren Abschluss hat den Abstand n zum aktuellen SP , während der letzte nur noch den Abstand 1 besitzt.

Beispiel 3.10.1 Betrachten wir den Ausdruck:

$$e \equiv \text{let rec } f = \text{fun } x \ y \rightarrow \text{if } y \leq 1 \text{ then } x \\ \text{else } f \ (x \cdot y) \ (y - 1) \\ \text{in } f1$$

in einer leeren Adressumgebung $\rho = \emptyset$ mit Kellerpegel $kp = 0$. Dann ergibt sich (für CBV):

0	alloc 1	0	$A :$	targ 2	4	loadc 1
1	pushloc 0	0		...	5	mkbasic
2	mkvec 1	1		return 2	5	pushloc 4
2	mkfunval A	2	$B :$	rewrite 1	6	apply
2	jump B	1		mark C	2	$C :$ slide 1

Den Code für den Rumpf von f haben wir zur Abkürzung weggelassen. \square

Damit haben wir alle Bestandteile beisammen, um die Programmiersprache FuL mit CBV-Semantik zu übersetzen und auszuführen. Zur Implementierung von CBN fehlt uns noch die Implementierung der Instruktion **eval** zur Auswertung von Abschlüssen – und natürlich die Übersetzungsfunktion `codeC` zur Erzeugung von Abschlüssen. Dies ist Gegenstand des nächsten Abschnitts.

3.11 Abschlüsse und ihre Auswertung

Abschlüsse werden zur Implementierung von CBN benötigt. Bei CBN können wir bei Zugriff auf eine Variable nicht sicher sein, dass deren Wert bereits vorliegt. Ist das nicht der Fall, müssen wir einen Kellerrahmen anlegen, innerhalb dessen der Wert ermittelt wird. Diese Aufgabe erledigt die Instruktion **eval**. Auch diese Instruktion lässt sich in übersichtlichere Bestandteile zerlegen:

```

eval = if ( $H[S[SP]] = (C, \_ \_)$ ) {
    mark0;                // Anlegen des Kellerrahmens
    pushloc 3;            // Kopieren des Verweises
    apply0;              // entspricht apply
}

```

Ein Abschluss kann aufgefasst werden als eine parameterlose Funktion, bei der folglich auf die *ap*-Komponente verzichtet werden kann. Auswerten des Abschlusses entspricht damit der Auswertung einer Anwendung einer solchen Funktion auf 0 Argumente. Damit kommen wir zu den einzelnen Schritten der Instruktion **eval**. Falls der Verweis oben auf dem Keller auf einen Abschluss zeigt, wird zuerst einmal ein Kellerrahmen angelegt. Dazu dient die Mikroinstruktion **mark0** (Abb. 3.26). Im Unterschied zu der uns bereits bekannten Instruktion **mark** *A* rettet **mark0** nicht eine feste Adresse *A* als Fortsetzungsadresse nach dem Aufruf, sondern den aktuellen *PC*, der ja gerade auf die erste Instruktion hinter **eval** zeigt.

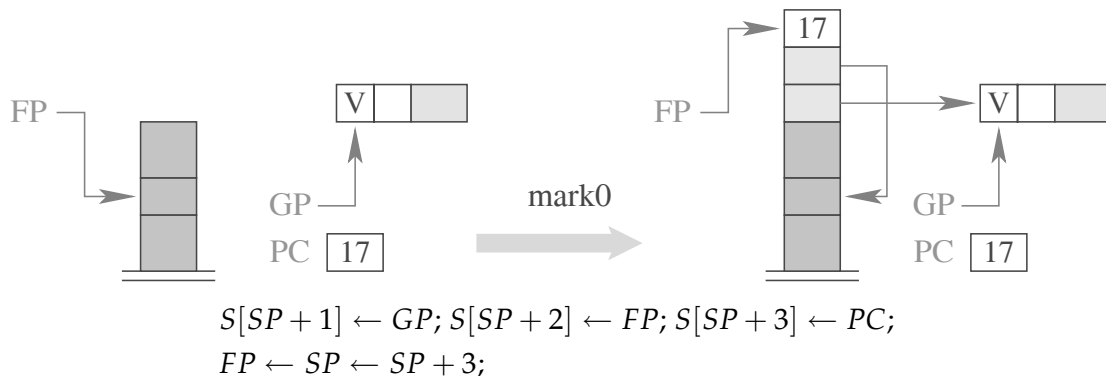
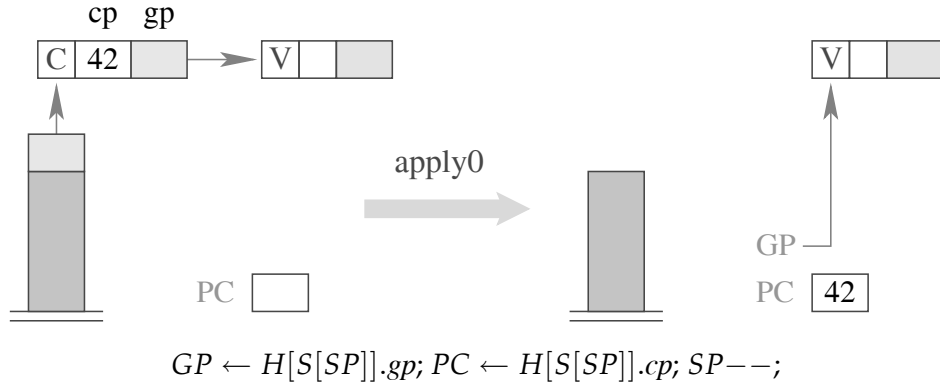
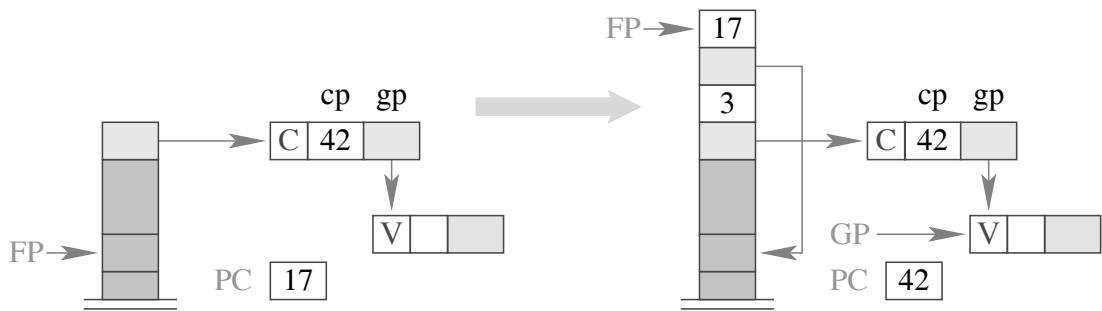


Abb. 3.26. Die Mikroinstruktion **mark0**.

Nach Anlegen des Kellerrahmens kopiert **pushloc** 3 den Verweis auf den Abschluss unterhalb des Kellerrahmens oben auf den Keller. Die Mikroinstruktion **apply0** (Abb. 3.27) packt den Abschluss oben auf dem Keller aus und springt den zugehörigen Code an. Diese Instruktion verhält sich wie die uns bereits bekannte Instruktion **apply** — mit dem Unterschied, dass ein Abschluss keinen *ap*-Vektor enthält, dessen Verweise zuerst auf den Keller gelegt werden müssten.

Den zusammengesetzten Effekt der Instruktion **eval** zeigt Abb. 3.28.

Abb. 3.27. Die Mikroinstruktion **apply0**.Abb. 3.28. Der Effekt der Instruktion **eval**.

Wenden wir uns nun der Herstellung von Abschlüssen zu. Die Herstellung eines Abschlusses für einen Ausdruck e erfordert erstens das Sammeln der Bindungen für die freien Variablen in einem Vektor, zweitens die Erzeugung eines C-Objekts, welches neben diesem Vektor einen Verweis auf den Code zur Berechnung des Werts von e enthält, gefolgt von der Überschreibung des Abschlusses mit dem berechneten Wert und Aufgabe des Kellerrahmens der Auswertung. Diesen Code liefert die Codeerzeugungsfunktion code_C :

```

codeC  $e \ \rho \ kp =$    getvar  $z_0 \ \rho \ kp$ 
                     getvar  $z_1 \ \rho \ (kp + 1)$ 
                     ...
                     getvar  $z_{g-1} \ \rho \ (kp + g - 1)$ 
                     mkvec  $g$ 
                     mkclos  $A$ 
                     jump  $B$ 
 $A : \text{code}_V \ e \ \rho' \ 0$ 
                     update
 $B : \dots$ 

```

wobei

$$\{z_0, \dots, z_{g-1}\} = \text{free}(e) \quad \text{und} \\ \rho' = \{z_i \mapsto (G, i) \mid i = 0, \dots, g-1\}$$

Den ersten Abschnitt des für den Ausdruck e erzeugten Codes kennen wir bereits von der Herstellung von F -Objekten, das Einpacken der Bindungen für die globalen Variablen in einen Vektor. Während wir dort mithilfe der Instruktion **mkfunval** A das F -Objekt herstellten, benutzen wir nun die neue Instruktion **mkclos** A , welche das entsprechende C -Objekt erzeugt (Abb. 3.29).

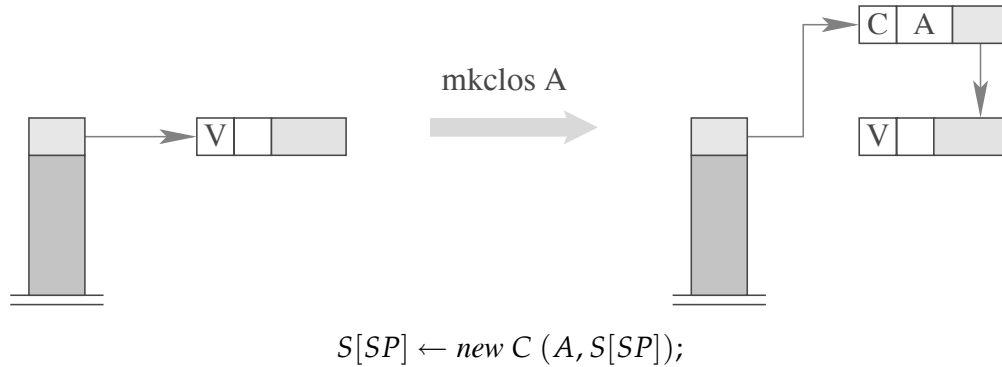


Abb. 3.29. Die Instruktion **mkclos** A .

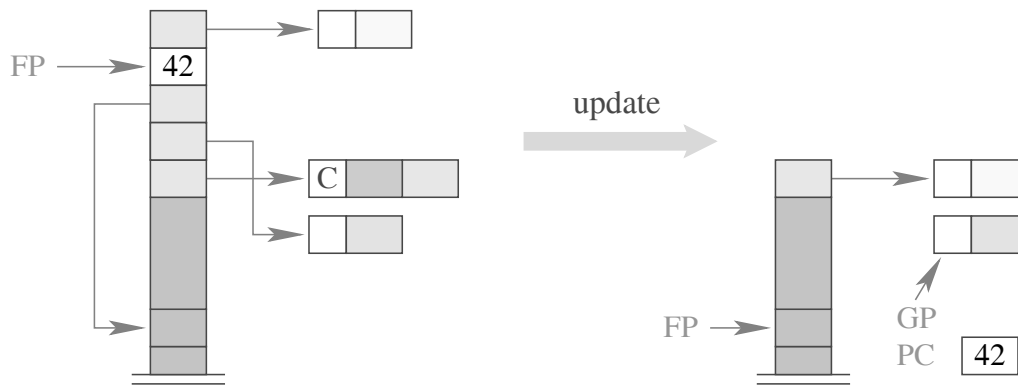
Beachten Sie, dass wir den Code zur Auswertung des Ausdrucks e direkt hinter den Code zur Erzeugung des Abschlusses platzieren. Insbesondere verwendet er für diesen die Adressumgebung ρ' , die sich aus der von uns gewählten Anordnung der globalen Variablen innerhalb des Global-Vektors ergibt. Bevor wir die Instruktion **update** diskutieren, schauen wir uns ein Beispiel an.

Beispiel 3.11.1 Betrachten wir den Ausdruck $e \equiv a * a$ in der Adressumgebung $\rho = \{a \mapsto (L, 0)\}$ bei Kellerpegel $kp = 1$. Dann liefert $\text{code}_C e \rho kp$ die Folge:

1	pushloc 1	0	$A :$	pushglob 0	2	getbasic
2	mkvec 1	1		eval	2	mul
2	mkclos A	1		getbasic	1	mkbasic
2	jump B	1		pushglob 0	1	update
		2		eval	2	$B :$...

In diesem Beispiel müssen wir nur die globale Variable a in einen Vektor einpacken. Der Code zur Auswertung des Ausdrucks steht in der zweiten und dritten Spalte. Ihm folgt die Instruktion **update**, mit deren Ausführung ein eventueller Aufruf von **eval** für den in der ersten Spalte erzeugten Abschluss endet. \square

Die Instruktion **update** soll einerseits den Kellerrahmen für die Auswertung des Ausdrucks aufgeben. Andererseits soll sie den Verweis auf den gerade ausgewerteten Abschluss mit dem berechneten Wert überschreiben. Damit entpuppt sie sich als Kombination der beiden Instruktionen (Abb. 3.30):

Abb. 3.30. Die Instruktion **update**.

update = **popenv**
 rewrite 1

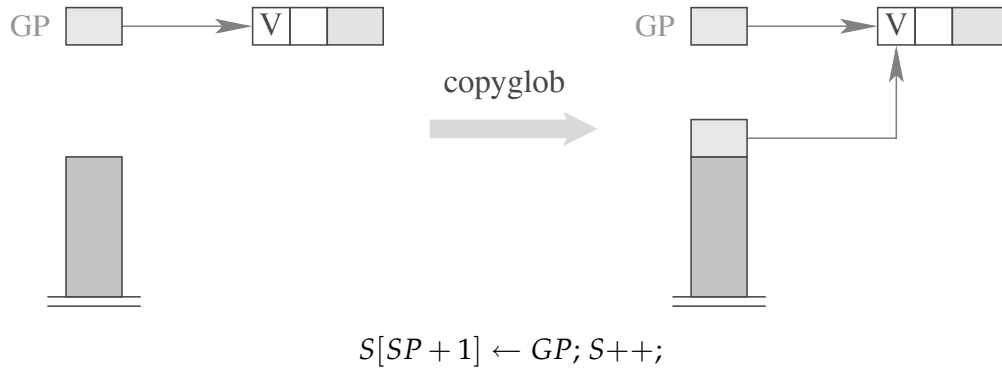
Damit ist die Übersetzung der funktionalen Kernsprache FuL sowohl bei CBN- wie bei CBN-Semantik abgeschlossen. Wir wollen jedoch nicht verschweigen, dass unsere Übersetzung erst einigermaßen effizienten Code liefert, wenn wir einige Optimierungen eingebaut haben.

3.12 Optimierungen I: Globale Variablen

Eine erste Optimierung betrifft die Behandlung globaler Variablen. In funktionalen Programmen werden typischerweise viele *F*- oder *C*-Objekte konstruiert. Dies erfordert jedesmal das Einpacken sämtlicher globaler Variablen des entsprechenden Ausdrucks. Diese Arbeit ließe sich verringern, wenn wir eine *Mehrfachverwendung* von bereits konstruierten Vektoren erlaubten! Dies ist sinnvoll etwa bei der Übersetzung von *let*- oder *letrec*-Ausdrücken oder auch bei der Konstruktion der Folge von Abschlüssen für die aktuellen Parameter eines Funktionsaufrufs. Die mehrfach benutzbaren Vektoren merken wir uns wie lokale Variablen im Kellerrahmen mit entsprechenden Relativadressen, welche wir in der Adressumgebung verwalten.

Diese Optimierung ist häufiger anwendbar, wenn wir auch Global-Vektoren gestatten, die *mehr* Komponenten enthalten als nur die Variablen, die im Ausdruck vorkommen. Dann erscheint es auch sinnvoll, einen Zugriff auf den aktuellen Global-Vektor zu gestatten, z.B. mit einer Instruktion **copyglob** (Abb. 3.31).

Der Vorteil dieser Optimierung ist, dass die Konstruktion von *F*- und *C*-Objekten dadurch *öfter* billiger wird. Der Nachteil ist jedoch, dass wir gelegentlich überflüssige Komponenten in Global-Vektoren einbauen. Diese behindern jedoch die frühzeitige Freigabe nicht mehr benötigter Haldenobjekte, ein sehr unangenehmer Effekt, der zu sogenannten *Space Leaks* führen kann, also einer Programmausführung, die fortgesetzt Speicher konsumiert, obwohl tatsächlich viel weniger Platz für die Berechnung erforderlich ist.

Abb. 3.31. Die Instruktion **copyglob**.

3.13 Optimierungen II: Abschlüsse

Eines der wesentlichen Probleme von Programmiersprachen mit CBN-Semantik ist, dass der Aufbau und die Verwaltung von Abschlüssen einen Aufwand erfordert, der oftmals überflüssig ist. Auf die Konstruktion eines Abschlusses für einen Ausdruck e kann man sicherlich verzichten, wenn die Berechnung des Werts von e nicht teurer ist als die Herstellung des Abschlusses selbst! Dies ist etwa der Fall, wenn der Ausdruck e eine Konstante ist. Darum definieren wir:

$$\text{code}_C b \rho kp = \text{code}_V b \rho kp = \text{loadc } b \\ \text{mkbasic}$$

Diese Folge ersetzt:

mkvec 0	jump B	mkbasic	$B : \dots$
mkclos A	$A :$ loadc b	update	

Ein weiterer Fall sind Variablenzugriffe. Variablen sind entweder an Werte oder bereits an C-Objekte gebunden. Eine erneute Konstruktion eines Abschlusses erscheint unsinnig. Deshalb setzen wir:

$$\text{code}_C x \rho kp = \text{getvar } x \rho kp$$

Diese eine Instruktion ersetzt die Instruktionsfolge:

getvar $x \rho kp$	mkclos A	$A :$ pushglob 0	update
mkvec 1	jump B	eval	$B : \dots$

Beispiel 3.13.1 Betrachten wir den Ausdruck:

let rec $a = b$ **and** $b = 7$ **in** a

Dann liefert bei CVN-Semantik $\text{code}_V e \emptyset 0$ mit unserer Optimierung die Folge:

0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

Die Ausführung dieser Folge sollte eigentlich den Basiswert 7 liefern. Tatsächlich können Sie sich jedoch davon überzeugen, dass die Auswertung zu einem Aufruf der Instruktion **eval** für einen Verweis auf einen leeren Abschluss führt.

Intuitiv liegt das daran, dass hier die Variable a initialisiert wird, indem ihr zugehöriger Dummy-Abschluss erneut durch einen leeren Abschluss überschrieben wird, nämlich denjenigen, den der Variablenzugriff auf b lieferte. Später wird zwar der leere Abschluss für b durch das B -Objekt $(B, 7)$ überschrieben; davon kann aber die Variable a nicht mehr profitieren. Offenbar liefert unsere Optimierung also nicht immer korrekten Code. \square

Das Problem mit unserer optimierten Codeerzeugung für Variablenzugriffe entsteht tatsächlich nur bei rekursiven Definitionen von Variablen durch Variablen, bei denen eine Variable y als rechte Seite verwendet wird, bevor der ihr zugehörige leere Abschluss überschrieben wurde.

Dieses Problem können wir glücklicherweise umgehen, indem wir erstens (offenbar unsinnige) zyklische Variablendefinitionen wie $y = y$ zur Übersetzungszeit zurückweisen und ansonsten die Definitionen $y_i = y_j$ so anordnen, dass der leere Abschluss für y_j stets vor dem leeren Abschluss für y_i überschrieben wird.

Auch für Funktionen e kann die Codeerzeugungsfunktion code_C verbessert werden: Funktionen sind bereits Werte, die nicht weiter evaluiert werden müssen. Anstatt Code zu erzeugen, der bei Bedarf ein F -Objekt erzeugt, können wir dieses auch direkt anlegen. Darum definieren wir:

$$\text{code}_C (\text{fun } x_0 \dots x_{k-1} \rightarrow e) \rho kp = \text{code}_V (\text{fun } x_0 \dots x_{k-1} \rightarrow e) \rho kp$$

Den Vergleich mit der entsprechenden unoptimierten Befehlsfolge überlassen wir Ihnen zur Übung.

Nach diesen Überlegungen komplettieren wir unser Programm der Codeerzeugung für funktionale Programme und erläutern, wie für ein ganzes Programm Code erzeugt wird.

3.14 Die Übersetzung eines Programmausdrucks

Die Ausführung für ein Programm e startet mit

$$PC = 0 \quad SP = FP = GP = -1$$

Der Ausdruck e sollte keine freien Variablen enthalten. Der Code, den wir für e erzeugen, sollte den Wert von e ermitteln und dann eine Instruktion **halt** ausführen:

```
code  $e = \text{code}_V e \ \emptyset \ 0$ 
halt
```

Damit ist die Codeerzeugung komplett. Es sollte jedoch nicht verschwiegen werden, dass die Code-Schemata, so wie wir sie bisher definiert haben, *Spaghetti-Code* liefern, ja, es wird die Anekdote kolportiert, dass jemand einen *obfuscated C contest* gewonnen habe, indem er eine Übersetzung eines funktionalen Programms nach C ablieferte.

In unserem Fall liegt der Grund darin, dass wir den Code zur Auswertung von Funktionsrümpfen und Abschlüssen stets direkt hinter den Befehlen **mkfunval** A bzw. **mkclos** A ablegen. Dieser Code könnte aber auch an einer beliebigen anderen Stelle im Programm stehen, also z.B. hinter dem **halt**-Befehl!

Eine solche Entwurfsentscheidung hätte den Vorteil, dass wir die direkten Sprünge über die Konstruktion von F - oder C -Objekten hinweg einsparen würden. Der Nachteil ist aber, dass unsere Codeerzeugungsfunktionen erheblich umständlicher würden, da sie zusätzlich einen *Codezwischenpeicher* verwalten müssten, in dem die ausgelagerten Programmstücke akkumuliert würden.

Wir verzichten deshalb auf ein solches Vorgehen und überlassen stattdessen die Sprungentwerrung einer separaten Optimierungsphase.

Beispiel 3.14.1 Betrachten wir das folgende Programm:

```
let  $a = 17$  in  $f = \text{fun } b \rightarrow a + b$  in  $f \ 42$ 
```

Entwerrung der Sprünge der Instruktionsfolge bei CBN-Semantik liefert:

0	loadc 17	2	mark B	3	$B :$ slide 2	1	pushloc 1
1	mkbasic	5	loadc 42	1	halt	2	eval
1	pushloc 0	6	mkbasic	0	$A :$ targ 1	2	getbasic
2	mkvec 1	6	pushloc 4	0	pushglob 0	2	add
2	mkfunval A	7	eval	1	eval	1	mkbasic
		7	apply	1	getbasic	1	return 1

□

3.15 Strukturierte Daten

Jede brauchbare funktionale Programmiersprache verfügt nicht nur über einfache Datentypen wie etwa ganze Zahlen und Boolesche Werte, sondern auch über strukturierte Datentypen wie Tupel und Listen. Wir beginnen mit der Einführung von *Tupeln*.

3.15.1 Tupel

Tupel werden mithilfe von Konstruktoren $(., \dots, .)$ der Stelligkeit ≥ 0 aufgebaut und mithilfe von *let*-Ausdrücken und *Pattern Matching* wieder zerlegt.

Entsprechend erweitern wir die Syntax unserer Programmausdrücke durch:

$$e ::= \dots \mid (e_0, \dots, e_{k-1}) \mid \\ \mathbf{let} (y_0, \dots, y_{k-1}) = e_1 \mathbf{in} e_0$$

für $k \geq 0$. Tupel werden angelegt, indem erst die Folge der Referenzen auf ihre Komponenten auf dem Keller gesammelt wird und diese Folge dann mithilfe der uns bereits bekannten Instruktion **mkvec** k in die Halde gelegt wird.

$$\begin{aligned} \text{code}_V (e_0, \dots, e_{k-1}) \rho \, kp &= \text{code}_C e_0 \rho \, kp \\ &\quad \text{code}_C e_1 \rho \, (kp + 1) \\ &\quad \dots \\ &\quad \text{code}_C e_{k-1} \rho \, (kp + k - 1) \\ &\quad \mathbf{mkvec} \, k \end{aligned}$$

Im Falle von CBV muss natürlich direkt Code zur Auswertung der Ausdrücke e_i erzeugt werden. Möchte man umgekehrt auf die Komponenten eines Tupels zugreifen, kann man dies mithilfe des Ausdrucks

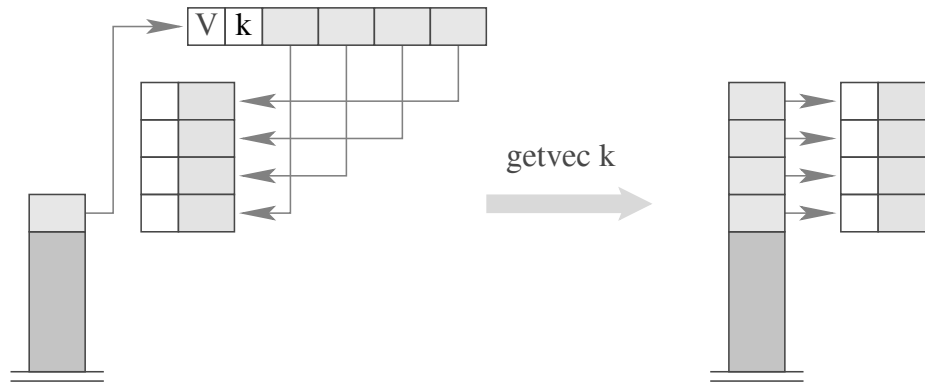
$$e \equiv \mathbf{let} (y_0, \dots, y_{k-1}) = e_1 \mathbf{in} e_0$$

tun. Die Idee besteht darin, den Wert von e_1 mit dem Tupelmuster (y_0, \dots, y_{k-1}) zu vergleichen und die im Muster vorkommenden Variablen y_i an die entsprechenden Komponenten des Werts zu binden. Diesen Vorgang nennt man auch *Pattern Matching*. Beachten Sie, dass wir hier nur sehr einfache Muster betrachten. Im Allgemeinen können Muster beliebig tief geschachtelt aus Konstanten, Variablen und Konstruktoren aufgebaut sein. In unserem einfachen Fall gehen wir bei der Implementierung so vor. Wir erzeugen für den Ausdruck e zuerst Code, der den Wert der rechten Seite e_1 ermittelt. Bei Programmausführung sollte dieser einen Vektor der Länge k auf der Halde und einen Verweis auf diesen Vektor auf dem Keller liefern. Die Verweise in diesem Vektor sollen nun auf den Keller geschoben werden, bevor wir uns an die Auswertung des Hauptausdrucks e_0 machen. Nach Abarbeitung des Hauptausdrucks kann der Platz für die k lokalen Variablen wieder freigegeben werden. Insgesamt ergibt sich das folgende Übersetzungsschema:

$$\begin{aligned} \text{code}_V e \rho \, kp &= \text{code}_V e_1 \rho \, kp \\ &\quad \mathbf{getvec} \, k \\ &\quad \text{code}_V e_0 \rho' \, (kp + k) \\ &\quad \mathbf{slide} \, k \end{aligned}$$

wobei $\rho' = \rho \oplus \{y_i \mapsto kp + i + 1 \mid i = 0, \dots, k - 1\}$ die neue Adressumgebung für den Hauptausdruck e_0 ist.

Für die Aufgabe, die Komponenten eines Vektors der Länge $k \geq 0$ auf den Keller zu legen, führen wir die neue Instruktion **getvec** k ein (Abb. 3.32). Unsere Implementierung des Befehls **getvec** k testet zusätzlich, dass der Argumentverweis oben



```

h ← S[SP];
if (H[h] ≠ (V, k, _)) error ("incompatible types");
for (i ← 0, i < k, i++) S[SP + i] ← H[h].v[i];
SP ← SP + k - 1;

```

Abb. 3.32. Die Instruktion **getvec k**.

auf dem Keller tatsächlich auf ein V -Objekt der richtigen Länge zeigt. Für statisch getypte Programmiersprachen wird (bei korrekter Codeerzeugung) dies allerdings stets der Fall sein.

3.15.2 Listen

Als Beispiel eines weiteren, nun *rekursiven* Datentyps betrachten wir Listen. Listen werden aus Listen-Elementen mithilfe der Konstante $[]$ (die leere Liste) und des rechts-assoziativen Operators „ $::$ “ (dem Listen-Konstruktor) aufgebaut. Im Unterschied zu Tupeln können Listen verschiedene Gestalt haben; der Vergleich mit Mustern kann deshalb auch zur *Fallunterscheidung* benutzt werden. Dazu dient der *match*-Ausdruck.

Beispiel 3.15.1 Betrachten wir etwa die Funktion *app*, die zwei Listen als Argumente nimmt und die zweite hinter die erste hängt:

```

app    =    fun l y → match l with
                []      →    y
                | h :: t →    h :: (app t y)

```

□

Darum erweitern wir die Syntax von Ausdrücken e um:

$$\begin{aligned}
 e ::= & \dots \mid [] \mid (e_1 :: e_2) \\
 & \mid (\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2)
 \end{aligned}$$

Im Allgemeinen können Muster natürlich beliebig tief geschachtelt sein. Dann wird eine Fallunterscheidung auch möglicherweise mehr als zwei Fälle zu unterscheiden

haben. Hier betrachten wir der Einfachheit halber aber nur Muster ohne Schachtelung von Konstruktoren.

Zur Implementierung von Listen benötigen wir neue Haldenobjekte (Abb. 3.33). Neben dem Etikett L , das sie als Listenknoten identifiziert, enthalten sie zusätzlich

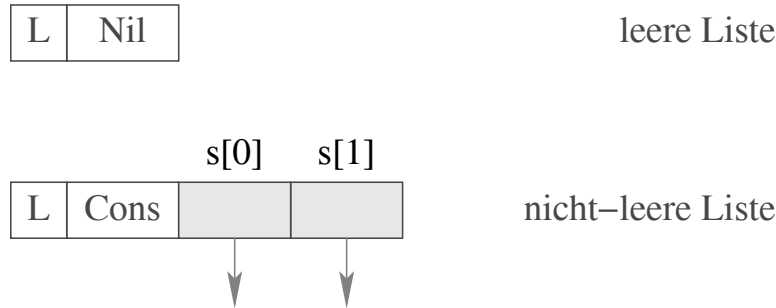


Abb. 3.33. Haldenobjekte für Listen.

den Namen des Konstruktors. Da Listen in funktionalen Sprachen sehr häufig benutzt werden, wird eine konkrete Implementierung natürlich für L -Objekte möglichst kompakte und effizient zu benutzende Spezialknoten verwenden.

Für das Anlegen von Listenknoten führen wir die Befehle **nil** (Abb. 3.34) und **cons** ein (Abb. 3.35). Der Aufruf $new\ L\ (Nil)$ soll ein L -Objekt für eine leere Li-

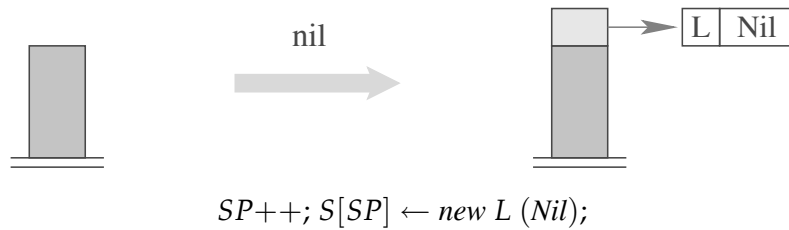
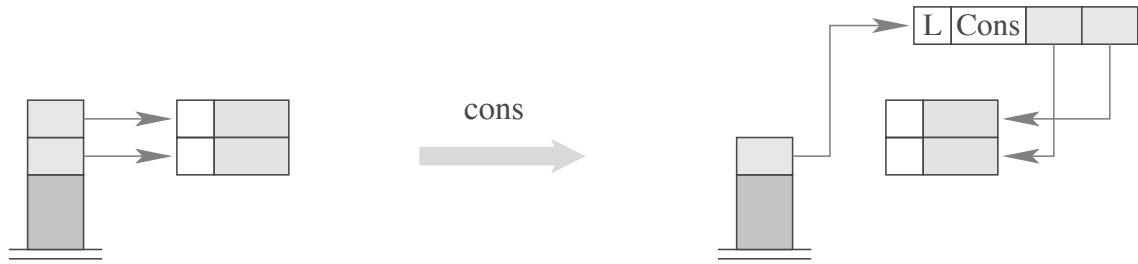


Abb. 3.34. Die Instruktion **nil**.

ste in der Halde erzeugen und einen Verweis darauf zurückliefern. Von dem Aufruf $new\ L\ (Cons, h, t)$ dagegen erwarten wir, dass er ein L -Objekt für eine zusammengesetzte Liste anlegt und darin die beiden Verweise h und t ablegt.

Mit den beiden neuen Instruktionen **nil** und **cons** übersetzen wir für CBN:

$$\begin{aligned} \text{code}_V [] \ \rho \ kp &= \mathbf{nil} \\ \text{code}_V (e_1 :: e_2) \ \rho \ kp &= \text{code}_C e_1 \ \rho \ kp \\ &\quad \text{code}_C e_2 \ \rho \ (kp + 1) \\ &\quad \mathbf{cons} \end{aligned}$$



$$SP \leftarrow SP + 1; S[SP] \leftarrow \text{new } L(\text{Cons}, S[SP], S[SP + 1]);$$

Abb. 3.35. Die Instruktion **cons**.

Im Falle von CBV werden wir für e_1 und e_2 wieder die Codeerzeugungsfunktion code_V verwenden.

Es verbleibt uns noch, Code für einen Ausdruck:

$$e \equiv \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2$$

zu erzeugen. Die Auswertung dieses Ausdrucks erfordert zuerst einmal die Auswertung von e_0 . Anschließend muss für den Wert von e_0 überprüft werden, ob er ein L -Objekt darstellt. Ist das der Fall, muss unterschieden werden, ob er eine leere oder eine nichtleere Liste repräsentiert. Im Falle einer leeren Liste muss der Ausdruck e_1 ausgewertet werden. Im Falle einer nichtleeren Liste dagegen müssen die beiden Verweise des L -Objekts als neue lokale Variablen oben auf den Keller gelegt werden, bevor der Ausdruck e_2 ausgewertet wird. Folglich erhalten wir (für CBN wie für CBV):

$$\begin{aligned} \text{code}_V e \rho kp = & \quad \text{code}_V e_0 \rho kp \\ & \text{tlist } A \\ & \text{code}_V e_1 \rho kp \\ & \text{jump } B \\ & A : \text{code}_V e_2 \rho' (kp + 2) \\ & \text{slide } 2 \\ & B : \dots \end{aligned}$$

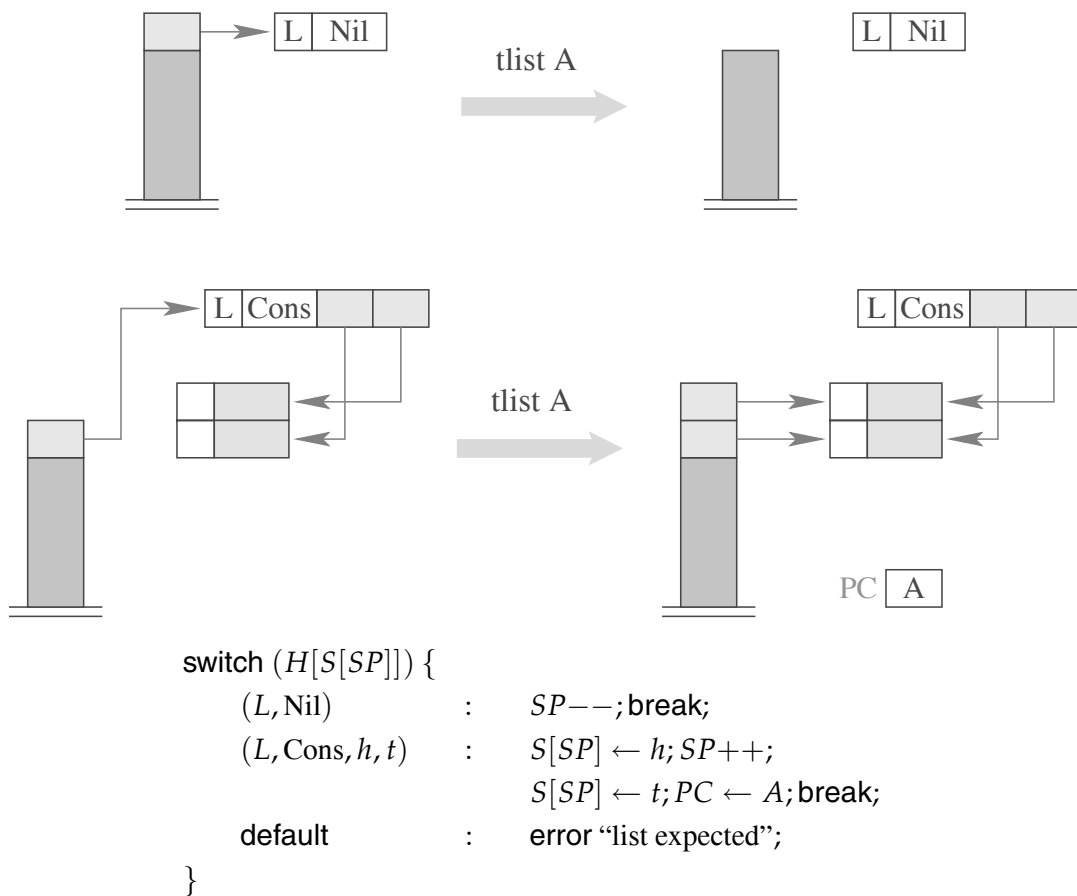
wobei die Adressumgebung für e_2 gegeben ist durch:

$$\rho' = \rho \oplus \{h \mapsto (L, kp + 1), t \mapsto (L, kp + 2)\}$$

Der neue Befehl **tlist** A führt die notwendigen Überprüfungen durch. Im *cons*-Fall legt er die beiden neuen lokalen Variablen für die Bestandteile der Liste an (Abb. 3.36) und schreibt die Adresse A in das Register PC . Beachten Sie, dass bei einem Datentyp mit mehr als zwei Konstruktoren wir sinnvollerweise einen *indizierten* Sprung über eine Sprungtabelle einsetzen (siehe Aufgabe 14).

Beispiel 3.15.2 Der (entwirrte) Rumpf der Funktion *app* für eine Adressumgebung ρ mit $\rho(\text{app}) = (G, 0)$ sieht so aus:

0	targ 2	3	pushglob 0	0	<i>C</i> :	mark <i>D</i>	
0	pushloc 0	4	pushloc 2	3		pushglob 2	
1	eval	5	pushloc 6	4		pushglob 1	
1	tlist <i>A</i>	6	mkvec 3	5		pushglob 0	
0	pushloc 1	4	mkclos <i>C</i>	6		eval	
1	eval	4	cons	6		apply	
1	jump <i>B</i>	3	slide 2	1	<i>D</i> :	update	
2	<i>A</i> :	pushloc 1	1	<i>B</i> :	return 2		□

Abb. 3.36. die Instruktion **tlist** *A*.

3.15.3 Abschlüsse für Tupel und Listen

Das generelle Schema für code_C lässt sich beim Anlegen von Tupeln und Listen optimieren, indem wir die entsprechenden Werte direkt aufbauen, anstatt ihren Aufbau zu verzögern:

$$\begin{aligned}
\text{code}_C (e_0, \dots, e_{k-1}) \rho kp &= \text{code}_C e_0 \rho kp \\
&\quad \text{code}_C e_1 \rho (kp + 1) \\
&\quad \dots \\
&\quad \text{code}_C e_{k-1} \rho (kp + k - 1) \\
&\quad \mathbf{mkvec} \ k \\
\text{code}_C [] \rho kp &= \mathbf{nil} \\
\text{code}_C (e_1 :: e_2) \rho kp &= \text{code}_C e_1 \rho kp \\
&\quad \text{code}_C e_2 \rho (kp + 1) \\
&\quad \mathbf{cons}
\end{aligned}$$

3.16 Optimierungen III: Letzte Aufrufe

Zum Schluss wollen wir noch eine Optimierung diskutieren, die – zumindest bei Programmiersprachen mit CBV-Semantik – entscheidend für die praktische Effizienz einer Implementierung ist: die Optimierung letzter Aufrufe. Diese Optimierung erlaubt, für endrekursive (tail recursive) Funktionen (fast) so effizienten Code zu erzeugen wie für Schleifen in imperativen Programmiersprachen.

Das Vorkommen eines Aufrufs $l \equiv e' e_0 \dots e_{m-1}$ heißt *letztes Vorkommen* in dem Ausdruck e , falls die Auswertung von l den Wert für e liefern kann.

Beispiel 3.16.1 Das Ausdrucksvorkommen $r \ t \ (h :: y)$ ist ein letztes in:

$$\mathbf{match} \ x \ \mathbf{with} \ [] \rightarrow y \mid h :: t \rightarrow r \ t \ (h :: y)$$

Das Vorkommen $f \ (x - 1)$ ist dagegen kein letztes in dem Ausdruck:

$$\mathbf{if} \ x \leq 1 \ \mathbf{then} \ 1 \ \mathbf{else} \ x \cdot f \ (x - 1)$$

Der Wert des Aufrufs muss erst noch mit x multipliziert werden, um den Wert des Gesamtausdrucks zu ergeben. \square

Offenbar gibt es ein rein syntaktisches Verfahren, um die Menge aller letzten Vorkommen von Aufrufen in einem Ausdruck zu ermitteln (Aufgabe 15).

Die wesentliche Beobachtung ist, dass letzte Vorkommen von Aufrufen eines Funktionsrumpfs keinen neuen Kellerrahmen benötigen! Stattdessen reicht es, die konsumierten Argumente vom Keller zu entfernen und mit dem Aufruf der Funktion e' fortzufahren.

Der optimierte Code für einen letzten Aufruf $l \equiv (e' e_0 \dots e_{m-1})$ in einer Funktion f mit k Argumenten muss deshalb die folgenden Tätigkeiten ausführen. Zunächst muss er die aktuellen Parameter e_i anlegen und die Funktion e' bestimmen. Dann sollte er die lokalen Variablen sowie die k verbrauchten Argumente von f freigeben. Schließlich sollte er die Instruktion **apply** ausführen.

Diese Überlegung führt uns zu dem folgenden Übersetzungsschema für einen letzten Aufruf l . Wir geben zuerst die Variante für CBN an:

```

codeV l ρ kp = codeC em-1 ρ kp
               codeC em-2 ρ (kp + 1)
               ...
               codeC e0 ρ (kp + m - 1)
               codeV e' ρ (kp + m)           // Auswerten der Funktion
               slide r (m + 1)                // Freigabe
               apply

```

wobei $r = kp + k$ die Anzahl der freizugebenden Kellerzellen ist. Beachten Sie, dass wir hier zum Verschieben der aktuellen Parameter sowie der Funktion des letzten Aufrufs auf die Instruktion **slide** a b zurückgreifen, die uns bereits bei der C-Maschine auf Seite 46 in Abb. 2.30 begegnet ist.

Beispiel 3.16.2 Der Rumpf der Funktion

$$r = \mathbf{fun} \ x \ y \rightarrow \mathbf{match} \ x \ \mathbf{with} \ [] \rightarrow y \mid h :: t \rightarrow r \ t \ (h :: y)$$

liefert dann:

0	targ 2	1		jump B	4		pushglob 0
0	pushloc 0				5		eval
1	eval	2	A :	pushloc 1	5		slide 4 3
1	tlist A	3		pushloc 4			apply
0	pushloc 1	4		cons			slide 2
1	eval	3		pushloc 1	1	B :	return 2

Da der alte Kellerrahmen beibehalten wird, wird **return** 2 nur über den direkten Sprung am Ende der $[]$ -Alternative erreicht. □

3.17 Aufgaben

1. *Funktionale Programmierung.* Schreiben Sie in OCaml eine Funktion, die eine Zahl als Eingabe nimmt und überprüft, ob es sich um eine Primzahl handelt. Das Ergebnis soll ein Boolescher Wert sein.
2. *Funktionale Programmierung.* Programmieren Sie die Fibonacci-Funktion mithilfe akkumulierender Parameter, so dass sie in linearer Zeit läuft!
3. *Funktionale Programmierung.* Gegeben sei das Programm:

```

let rec x = 1
and f = fun y → x + y + z
and z = 4

```

```

in let rec x = 2
  and g = fun y → x + (f y) + z
  in let rec x = 3
    and f = fun y → y
    and z = 5
    in g 6

```

Was ist die Ausgabe des Programms

- bei statischer Bindung, bzw.
 - bei dynamischer Bindung?
4. *Funktionale Programmierung.* Implementieren Sie die folgenden Funktionen in OCaml (alternativ in SML) ohne Verwendung entsprechender Bibliotheken:
- member* nimmt als Argumente ein Element e und eine Liste l und liefert zurück, ob das Element e in der Liste l enthalten ist.
 - filter* nimmt als Argumente ein Prädikat p und eine Liste l und liefert eine Liste der Elemente aus l , für die p wahr ist.
 - fold* fasst sukzessive die Elemente der Liste mit der Funktion f zusammen. D.h. es gilt:

$$\text{fold } f \ e \ [x_1; x_2; \dots; x_n] = f \ (\dots (f \ (f \ e \ x_1) \ x_2) \ \dots) \ x_n$$

- mapi* nimmt als Argumente eine Funktion f und eine Liste l und wendet f auf jedes Paar eines Elements aus l und dessen Position in l an:

$$\text{mapi } f \ [x_1; x_2; \dots; x_n] = [f \ x_1 \ 1; f \ x_2 \ 2; \dots; f \ x_n \ n]$$

Für $f \ x \ i = x + i$ soll z.B. $\text{mapi } f \ [3; 3; 3]$ die Liste $[4; 5; 6]$ liefern.

5. *Freie Variablen.* Bestimmen Sie die Mengen der freien Variablen in den folgenden FuL-Ausdrücken!

```

(fun x → x y) (fun y → y)
fun x y → z (fun z → z(fun x → y))
(fun x y → x z (y z)) (fun x → y(fun y → y))
fun x → x + let rec a = x
  and x = f y
  and y = z
  in x + y + z

```

6. *Übersetzung und Kellerpegel.* Gegeben sei der Ausdruck

$$e \equiv \text{if } x > 1 \text{ then } x \text{ else let } z = x + y \text{ in } z + z$$

Berechnen Sie $\text{code}_v \ e \ \rho \ kp$ für die Adressumgebung $\rho = \{x \mapsto (L, 1), y \mapsto (L, -1)\}$ und den Kellerpegel $kp = 3$. Geben Sie analog zu den Beispielen im Text für jede Instruktion den aktuellen Kellerpegel an.

7. *Übersetzung von Funktionen.* Gegeben sei ein Ausdruck:

$$e \equiv \text{fun } x \ y \rightarrow \text{if } x = 1 \text{ then } y \text{ else } fac \ (x - 1) \ (x \cdot y)$$

Berechnen Sie $\text{code}_V e \ \rho \ kp$ für die Adressumgebung $\rho = \{fac \mapsto (L, 1)\}$ und den Kellerpegel $kp = 5$.

8. *Adressierung von Variablen.* Führen Sie ein neues Register SP_0 ein, relativ dem die lokalen Variablen adressiert werden können. Führen Sie dazu einen neuen Befehl für den Zugriff auf lokale Variablen ein! Wie müssen Sie die MaMa-Codeerzeugung modifizieren, dass dieses zusätzliche Register korrekt verwaltet wird?

9. *Funktionen mit lokalen Variablen.* Gegeben sei die Funktionsdefinition:

```
fun x, y, z → let x = 1
              in let a = 3
              in let b = 4
              in (a + (b + (x + (y + z))))
```

- Bestimmen Sie die Adressen der Variablen a , b , x , y , und z relativ zu SP_0 (dem Kellerzeiger beim Betreten des Funktionsrumpfes).
 - Bestimmen Sie die Adressumgebung ρ , die zur Code-Generierung von $(a + (b + (x + (y + z))))$ verwendet wird!
 - Erzeugen Sie Call-by-value-Code für die Funktionsdefinition! Geben Sie zu jeder Instruktion den aktuellen Kellerpegel an. Der Ausgangskellerpegel sei $kp = 3$.
10. *Reverse Engineering.* Betrachten Sie folgenden Call-by-value-Code eines FuL-Ausdrucks:

alloc 1	getbasic	jumpz _5	
pushloc 0	pushloc 5	mark _7	
mkvec 1	getbasic	pushloc 4	_3 : return 2
mkfunval _0	sub	getbasic	_1 : rewrite 1
jump _1	mkbasic	pushloc 4	mark _8
_0 : targ 2	pushloc 5	getbasic	loadc 16
pushloc 0	pushglob 0	sub	mkbasic
getbasic	apply	mkbasic	loadc 12
pushloc 2	_4 : jump _3	pushloc 4	mkbasic
getbasic	_2 : pushloc 0	pushglob 0	pushloc 5
gr	getbasic	apply	apply
jumpz _2	pushloc 2	_7 : jump _6	_8 : slide 1
mark _4	getbasic	_5 : pushloc 0	halt
pushloc 3	le	_6 :	

- Bestimmen Sie den Kellerpegel kp für jeden Programmpunkt beginnend bei $kp = 0$!

- Was berechnet dieses Programm? Versuchen Sie hierzu den Code in Sinn-einheiten zu zerlegen.
11. *Übersetzung ganzer Programme.* Erzeugen Sie Code für folgendes Programm (CBV und CBN):

```

let rec fib =
    if x < 3 then x
    else fib (x - 1) + fib (x - 2)
in fib 7

```

12. *Code-Optimierung für Call-by-Need.* Die Codeerzeugungsfunktion nach der CBN-Strategie generiert bei jedem Variablenzugriff eine *eval*-Instruktion. Falls die Variable bereits ausgewertet wurde, ist die *eval*-Instruktion jedoch überflüssig. So ist bei der Übersetzung des Ausdrucks $(a + a + a)$ eine *eval*-Instruktion nur nach dem ersten Vorkommen der Variablen a notwendig. Durch Erweiterung der Codeerzeugungsfunktion um ein weiteres Argument A , welches die Menge der bereits ausgewerteten Variablen enthält, lassen sich überflüssige *eval*-Instruktionen einsparen. Für zweistellige Operatoren haben wir:

$$\begin{aligned}
 \text{code}_V (e_1 \square_2 e_2) \rho kp A &= \text{code}_B e_1 \rho kp A \\
 &\quad \text{code}_B e_2 \rho (kp + 1) (A \cup A[e_1]) \\
 &\quad \text{op}_2 \\
 &\quad \text{mkbasic} \\
 A[e_1 \square_2 e_2] &= A[e_1] \cup A[e_2]
 \end{aligned}$$

$A[e]$ ist dabei die Menge der Variablen in e , die sicher ausgewertet werden müssen, um den Wert von e_1 zu berechnen.

Die Codeerzeugungsfunktion für Variablenzugriffe lautet dann:

$$\text{code}_V x \rho kp A = \begin{cases} \text{getvar } x \rho kp & , \text{ falls } x \notin A \\ \text{eval} & \\ \text{getvar } x \rho kp & , \text{ falls } x \in A \end{cases}$$

- Implementieren Sie $A[e]$ für beliebige FuL-Ausdrücke e .
 - Modifizieren Sie die Codeerzeugungsfunktion für FuL-Ausdrücke, um überflüssige *eval*-Instruktionen einzusparen.
13. *Referenzen und Seiteneffekte.* Wir erweitern FuL mit CBV-Semantik um Referenzen mit der folgende Syntax:

$$e ::= \dots \mid () \mid \mathbf{ref} \ e \mid e_1 \leftarrow e_2 \mid !e \mid e_1; e_2$$

$()$ ist ein nullstelliger Wertkonstruktor (lies: unit). Der Wert von $\mathbf{ref} \ e$ ist ein neues Haldenobjekt, das (einen Verweis auf) e enthält. Der Wert von $e_1 \leftarrow e_2$ ist $()$.

Allerdings wird vor Rückgabe von $()$ die linke Seite e_1 zu einem Referenzobjekt ausgewertet, in dem der Wert der rechten Seite e_2 abgelegt wird. Der Operator $!$ ist der Dereferenzierungsoperator, der den Wert, der in einem Referenz-Objekt gespeichert ist, zurückgibt. Ein $;$ -Ausdruck liefert den Wert des zweiten Arguments zurück, nachdem vorher das erste Argument ausgewertet wurde.

- (a) Erweitern Sie FuL um Referenzen. Sie müssen dazu neue Haldenobjekte für $()$ und Referenzen einführen. Gegebenenfalls können Sie auch neue Maschinenbefehle spezifizieren. Letztlich sollten Sie Code-Schemata für die Übersetzung der neuen Ausdrücke definieren.
 - (b) Der $;$ -Operator hat auch ohne Referenzen eine semantische Bedeutung. Ist er dann auch sinnvoll?
14. *Bäume.* Erweitern Sie FuL um den Typ **Tree**. **Trees** werden aus *Tree*-Elementen mithilfe der Konstante **Leaf** und den Konstruktoren **Node1**, **Node2** und **Node3** aufgebaut. Die Konstruktoren bauen einen *Tree*-Wert aus einem beliebigen Wert und einem, zwei oder drei *Tree*-Werten zusammen. Dazu erweitern wir die Syntax von Ausdrücken e um:

$$\begin{aligned}
 e ::= & \dots \mid \mathbf{Leaf} \mid \mathbf{Node1}(e_1, e_2) \\
 & \mid \mathbf{Node2}(e_1, e_2, e_3) \mid \mathbf{Node3}(e_1, e_2, e_3, e_4) \\
 & \mid (\mathbf{match} \ e_0 \ \mathbf{with} \ \mathbf{Leaf} \rightarrow e_1 \\
 & \qquad \qquad \qquad \mid \mathbf{Node1}(x, a_1) \rightarrow e_2 \\
 & \qquad \qquad \qquad \mid \mathbf{Node2}(x, a_1, a_2) \rightarrow e_3 \\
 & \qquad \qquad \qquad \mid \mathbf{Node3}(x, a_1, a_2, a_3) \rightarrow e_4)
 \end{aligned}$$

Definieren Sie die Codeerzeugungsfunktionen für die neu eingeführten Ausdrücke. Führen Sie dazu um neue Haldenobjekte vom Typ **Tree** ein.

15. *Letzte Aufrufe.* Zur Durchführung von Optimierungen letzter Aufrufe bei der Übersetzung eines Ausdrucks e müssen die Vorkommen letzter Aufrufe in e bestimmt werden. Das Aufruf-Vorkommen $l \equiv e'e_0 \dots e_{m-1}$ heißt *letzt* in einem Ausdruck e , falls die Auswertung von l den Wert für e liefern kann. Geben Sie ein allgemeines Schema für die Berechnung der letzten Aufruf-Vorkommen in einem Ausdruck e an!

3.18 Liste der Register der MaMa

FP , Rahmenzeiger (Frame Pointer)	S. 66
GP , Global-Zeiger (Global Pointer)	S. 71
PC , Befehlszähler (Program Counter)	S. 66
SP , Kellerzeiger (Stack Pointer)	S. 66

3.19 Liste der Codeerzeugungsfunktionen der MaMa

code_V	S. 67	code_B	S. 69
code_C	S. 68	code	S. 94

3.20 Liste der MaMa-Instruktionen

alloc	S. 87	mark	S. 80	pushglob	S. 74
add	S. 95	mark0	S. 89	pushloc	S. 74
apply	S. 81	mul	S. 69	return	S. 84
apply0	S. 89	mkbasic	S. 70	rewrite	S. 87
cons	S. 98	mkclos	S. 91	targ	S. 82
copyglob	S. 92	mkfunval	S. 78	popenv	S. 83
eval	S. 89	mkvec	S. 77	slide	S. 102
getbasic	S. 69	mkvec0	S. 83	targ	S. 82
getvec	S. 96	neg	S. 69	update	S. 91
halt	S. 66	nil	S. 98	wrap	S. 83
leq	S. 69				

3.21 Literaturhinweise

Eine der ersten abstrakten Maschinen zur Übersetzung funktionaler Programmiersprachen war die SECD-Maschine von Landin [Lan64]. Sie wurde entwickelt, um die Semantik für Lisp-Programme mit statischer Bindung und CBV-Parameterübergabe zu definieren. Die G-Maschine ist die erste abstrakte Maschine mit programmierter Graphreduktion. Sie wurde von Th. Johnsson [Joh84] eingeführt. Die G-Maschine und die TIM (Three Instruction Maschine) [FW87] haben den Entwurf der MaMa beeinflusst. Einen Überblick über die Implementierung funktionaler Programmiersprachen gibt [PJ87]. Die Weiterentwicklung der G-Maschine zur *Spineless Tagless G-Machine* bildet die Grundlage des Glasgow Haskell Compilers [Jon92]. Auf der virtuellen Maschine ZINC dagegen bauen die Übersetzer für CamlLight, MoscowML und OCaml auf [Ler90].

Die Übersetzung logischer Programmiersprachen

Die Idee der logischen Programmierung geht auf R. Kowalski und A. Colmerauer zurück, die Anfang der 70er Jahre entdeckten, dass man Ausdrücken der Prädikatenlogik eine operationale Interpretation geben kann. Dazu benutzt man die von J.A. Robinson 1965 für mechanische Beweisverfahren vorgeschlagene Resolutionsmethode als Berechnungsmodell.

Die *Klausel-Notation* ist ein besonders einfaches Format für universell quantifizierte prädikatenlogische Formeln der ersten Stufe. Für solche Formeln erlaubt die Resolutionsmethode die mechanisierte Herleitung eines Widerspruchs. Eine logische Formel α folgt genau dann aus einer Menge von Formeln S , wenn $S \cup \{\neg\alpha\}$ widerspruchsvoll ist. Deshalb eignet sich Resolution auch, um Folgerungsbeziehungen herzuleiten. Besonders einfach gestaltet sich Resolution für *Horn-Klauseln*. Eine Horn-Klausel formalisiert eine Regel, wie eine unmittelbare Folgerung aus einer endlichen Menge von Voraussetzungen gezogen werden kann. Die grundlegende Beobachtung ist, dass ein Resolutionsschritt für Horn-Klauseln ganz ähnlich auszuführen ist wie ein Prozeduraufruf.

Über logische Programme wird deshalb in drei verschiedenen Terminologien gesprochen. Geht es um Programmierung, so spricht man von Regeln, Prozeduren, Alternativen einer Prozedur, Aufrufen, Variablen usw. Erklärt man die prädikatenlogischen Grundlagen, so benutzt man Bezeichnungen wie Variablen-, Funktions- und Prädikatssymbol, Term, atomare Formel usw. Aus der Mechanisierung der Prädikatenlogik in automatisierten Beweisverfahren schließlich stammen die Begriffe wie Literal, Horn-Klausel, Unifikation und Resolution.

4.1 Die Sprache ProL

Im Folgenden führen wir die logische Programmiersprache ein, die wir übersetzen wollen. Um die Prinzipien der Übersetzung logischer Programmiersprachen zu erklären, beschränken wir uns auf eine Teilmenge der realen Programmiersprache Prolog. Wir wollen diese Kernsprache ProL nennen (**Pro**log **L**anguage). In dieser Sprache verzichten wir unter anderem auf die folgenden Konzepte von Prolog:

- Arithmetik,
- den Cut-Operator, sowie
- Meta-Programmierung und Selbst-Modifikation von Programmen etwa durch Hinzufügen oder Entfernen von Klauseln.

Da ein praktisches Programmieren in Prolog ohne Cut so gut wie unmöglich ist, werden wir eine Implementierung dieses Operators später nachtragen. Arithmetik behandeln wir in Aufgabe 10.

Ein ProL-Programm besteht aus einer Menge von Fakten und Regeln, zusammen mit einer Anfrage. Die Fakten und Regeln definieren Prädikate. Die Anfrage fragt, ob eine bestimmte Aussage erfüllbar ist, d.h. für mindestens eine Variablenbelegung wahr ist. Betrachten Sie z.B. das Prädikat *bigger* der Stelligkeit 2, das durch die folgenden Fakten definiert ist:

```
bigger(elephant, horse)  ⇐
bigger(horse, donkey)   ⇐
bigger(donkey, dog)     ⇐
bigger(donkey, monkey) ⇐
```

Mithilfe dieses Prädikats können wir leicht ein Prädikat definieren, welches den transitiven Abschluss von *bigger*-Beziehungen hinzufügt:

```
is_bigger(X, Y) ⇐ bigger(X, Y)
is_bigger(X, Y) ⇐ bigger(X, Z), is_bigger(Z, Y)
```

Die Namen *X*, *Y* und *Z* fungieren dabei als Variablen. Wie bei Prolog vereinbaren wir die Konvention, dass Variablen stets groß geschrieben werden (oder mit einem „_“ beginnen).

Zur Kompletierung unseres Programms benötigen wir noch eine Anfrage, etwa:

```
⇐ is_bigger(elephant, dog)
```

In diesem Fall enthält die Anfrage keine Variable. Die Antwort ist deshalb entweder **yes**, falls sich die Aussage mithilfe der Regeln und Fakten herleiten lässt, oder **no**, falls dies nicht der Fall ist. Hätten wir dagegen gefragt:

```
⇐ is_bigger(X, dog)
```

wären wir an möglichen Belegungen für die Variable *X* interessiert, für welche die Aussage abgeleitet werden kann. Bemerken Sie schon jetzt, dass hier offenbar mehrere Antworten möglich sind.

Um das Format von Regeln und Fakten weiter zu vereinfachen, nehmen wir in ProL zusätzlich an, dass auf der linken Seite der Implikation jeweils nur ein Prädikat, angewendet auf eine Liste paarweise verschiedener Variablen steht. Die eventuell bereits bekannten Bindungen für diese Variablen können wir in expliziten (Unifikations-) Gleichungen im Rumpf, d.h. auf der rechten Seite der Implikation definieren. Als ProL-Programm sieht unser Beispiel mit der zweiten Anfrage deshalb so aus:

$$\begin{aligned}
\text{bigger}(X, Y) &\Leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) &\Leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) &\Leftarrow X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) &\Leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is_bigger}(X, Y) &\Leftarrow \text{bigger}(X, Y) \\
\text{is_bigger}(X, Y) &\Leftarrow \text{bigger}(X, Z), \text{is_bigger}(Z, Y) \\
&\Leftarrow \text{is_bigger}(X, \text{dog})
\end{aligned}$$

Betrachten wir als weiteres, etwas realistischeres Beispiel das dreistellige Prädikat `app`, das die Konkatination zweier Listen beschreibt:

$$\begin{aligned}
\text{app}(X, Y, Z) &\Leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) &\Leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \\
&\Leftarrow \text{app}(X, [Y, c], [a, b, Z])
\end{aligned}$$

Die erste Regel besagt, dass die Liste Z die Konkatination der Listen X und Y ist, falls X gleich der leeren Liste und Y gleich Z ist. Die zweite Regel besagt, dass die Liste Z auch die Konkatination der Listen X und Y sein kann, falls X und Z als erste Elemente H enthalten und die Konkatination des Rests X' von X zusammen mit Y den Rest Z' von Z ergibt. Beachten Sie hier, dass in logischen Programmiersprachen eine etwas andere Syntax zur Repräsentation von Listen üblich ist: Kopf und Rest werden hier mithilfe des Infix-Operators „|“ getrennt.

Für eine Liste aus den Elementen X_1, \dots, X_n verwenden wir dabei gelegentlich die Kurzschreibweise $[X_1, \dots, X_n]$. Im Unterschied zu OCaml werden die einzelnen Listenelemente nicht durch „;“ sondern durch „|“ abgetrennt. Damit bezeichnet also z.B. $[]$ die leere Liste, $[H|Z']$ ist die Anwendung des Listenkonstruktors auf H und Z' , und $[a, b, Z]$ ist die Abkürzung für $[a|[b|[Z|[]]]]$.

Offenbar erlaubt uns die Programmiersprache ProL nicht nur, nach dem Ergebnis einer Konkatination zu fragen. Die verschiedenen Argumente des Prädikats sind vielmehr (zumindest prinzipiell) gleichberechtigt. Deshalb können wir bestimmte Vorgaben an die Argumente wie das Ergebnis einer Konkatination zweier Listen formulieren und es der Auswertung des ProL-Programms überlassen, Variablenbelegungen zu liefern, die diese Vorgaben erfüllen.

Ein ProL-Programm p ist darum wie folgt aufgebaut:

$$\begin{aligned}
t &::= a \mid X \mid _ \mid f(t_1, \dots, t_n) \\
g &::= q(t_1, \dots, t_k) \mid X = t \\
c &::= q(X_1, \dots, X_k) \Leftarrow g_1, \dots, g_r \\
a &::= \Leftarrow g_1, \dots, g_r \\
p &::= c_1 \dots c_m \quad a
\end{aligned}$$

Ein Term t ist damit entweder ein Atom a , eine Variable X , eine *anonyme* Variable $_$ oder die Anwendung eines Konstruktors f/n auf Terme t_1, \dots, t_n . Beachten

Sie, dass entsprechend dieser Konvention die *Stelligkeit* ein wesentlicher Bestandteil eines Konstruktors ist.

Ein *Ziel* g ist entweder ein *Literal*, d.h. ein Aufruf eines Prädikats q/k für Argumentterme t_1, \dots, t_k oder eine *Unifikation* $X = t$ wobei die linke Seite stets eine Variable X und die rechte Seite ein Term t ist.

Eine *Klausel* c besteht aus einem *Kopf* $q(X_1, \dots, X_k)$ mit Prädikat q/k und einer Liste von (paarweise verschiedenen) *formalen Parametern* sowie einer Folge von Zielen als *Rumpf*.

Eine *Anfrage* besteht aus einer Folge von Zielen. Ein *Programm* schließlich besteht aus einer Folge von Klauseln, gefolgt von einer Anfrage.

Die Idee zur effizienten Implementierung von ProL-Programmen besteht darin, die logische Sicht auf ProL-Programme durch eine prozedurale Sicht zu ersetzen. Aus dieser Perspektive fassen wir Prädikate als *Prozeduren* auf, die mit mithilfe der zugehörigen Fakten und Regeln definiert werden. Jede dieser Alternativen stellt somit einen möglichen Rumpf für die Prozedur dar, zwischen denen wir während der Programmausführung auswählen können. Entsprechend fassen wir Literale im Rumpf einer Regel oder in der Anfrage als *Prozeduraufrufe* auf. Beachten Sie, dass diese Aufrufe tatsächlich keinen Rückgabewert liefern.

Die *Werte*, mit denen unser Programm rechnet, sind Terme mit oder ohne freie Variablen. Die Schlüsseloperation während der Programmausführung ist die *Unifikation* zwischen solchen Termen. Diese Operation kann allerdings fehlschlagen: eine der Auswahlen von Fakten oder Regeln, die bisher getroffen wurde, führte in eine Sackgasse. Man kehrt dann zu einer solchen Stelle zurück, um dort eine andere Alternative auszuwählen. Das nennt man auch *Rücksetzen* oder *Backtracking*. Ist die Unifikation dagegen erfolgreich, so bewirkt sie die Bindung einiger bisher ungebundener Variablen als *Seiteneffekt*.

Im Folgenden präsentieren wir eine virtuelle Maschine für ProL. Unser Entwurf strebt Einfachheit und möglichst weitgehende Ähnlichkeit mit den bisherigen Maschinen, der C-Maschine für C und der MaMa für FuL an. Später werden einige Verbesserungen des Entwurfs vorgestellt, die die WiM (fast) brauchbar machen. Weitere effizienzsteigernde Maßnahmen, insbesondere solche, die auf dem Einsatz (virtueller) Register beruhen, werden wir hier nicht behandeln. Wie üblich, beginnen wir mit einer Diskussion der grundlegenden Architektur.

4.2 Die Architektur der WiM

Wie die C-Maschine oder die MaMa verfügt die WiM über einen *Programmspeicher* C sowie ein Register PC , den *Befehlszähler*, der auf die jeweils aktuelle Instruktion zeigt (Abb. 4.1). Wieder verwenden wir die Konvention, dass jede Zelle im Programmspeicher einen virtuellen Befehl aufnehmen kann. Der Hauptzyklus der WiM sieht genauso aus wie der der C-Maschine und der der MaMa:

- Laden des Befehls in $C[PC]$;
- Erhöhung des PC um 1;
- Ausführung des geladenen Befehls.

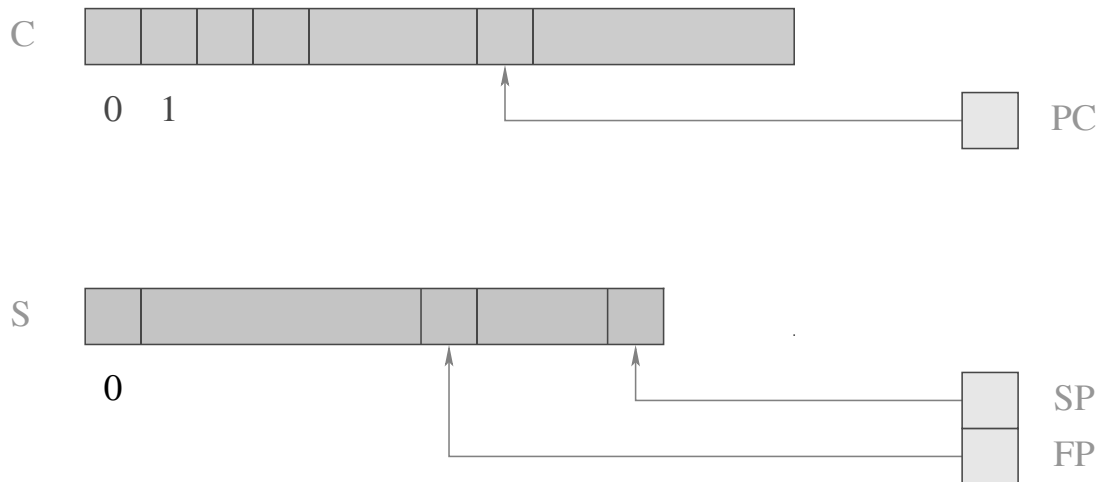


Abb. 4.1. Programmspeicher und Keller der WiM.

Zur Implementierung rekursiver Aufrufe wie für lokale Berechnungen benötigen wir in der WiM selbstverständlich auch einen *Keller S*, (vorerst) mit den Registern *SP* und *FP*, dem Kellerzeiger (Stack Pointer) und dem Rahmenzeiger (Frame Pointer). Das Register *SP* zeigt stets auf die oberste belegte Kellerzelle, während das Register *FP* auf den Kellerrahmen zeigt, in dem die lokalen Variablen des aktuellen Prozeduraufrufs liegen. Die einzelnen Zellen von *S* sollten in der Lage sein, einzelne Datenelemente, d.h. hier insbesondere Adressen aufzunehmen.

Wieder benötigen wir eine Datenstruktur *H*, die *Halde*, in der Repräsentationen der Werte verwaltet werden (Abb. 4.2). Im Unterschied zur Halde der virtuellen Maschine MaMa sind wir nun wieder etwas konkreter bzgl. der Implementierung dieser Datenstruktur. Insbesondere werden die Elemente in der Halde nach Möglichkeit ebenfalls *kellerartig* verwaltet. Dazu verwalten wir ein Register *HP*, den *Haldenzei-*

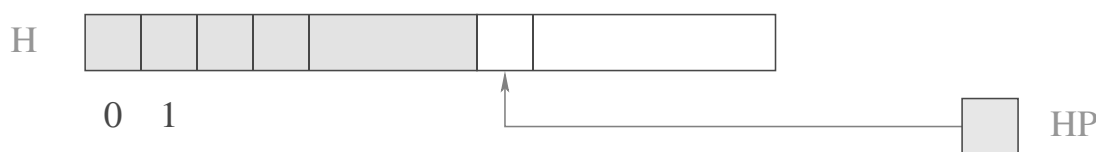


Abb. 4.2. Die Halde der WiM.

ger (Heap Pointer), der jeweils auf die erste freie Zelle zeigt. Die Idee hinter dieser konkreten Organisation ist, dass durch die kellerartige Verwaltung bei einem Rücksetzen der Berechnung alle zwischenzeitlich allokierten Objekte in einem Schritt wieder freigegeben werden können.

Die Haldenobjekte der WiM sind in der Abb. 4.3 zusammengefasst. Wie bei der MaMa sind die Objekte jeweils mit ihrem Typ markiert. Wir unterscheiden *A*-Objekte für atomare Terme oder Atome, *S*-Objekte für Strukturen und *R*-Objekte für

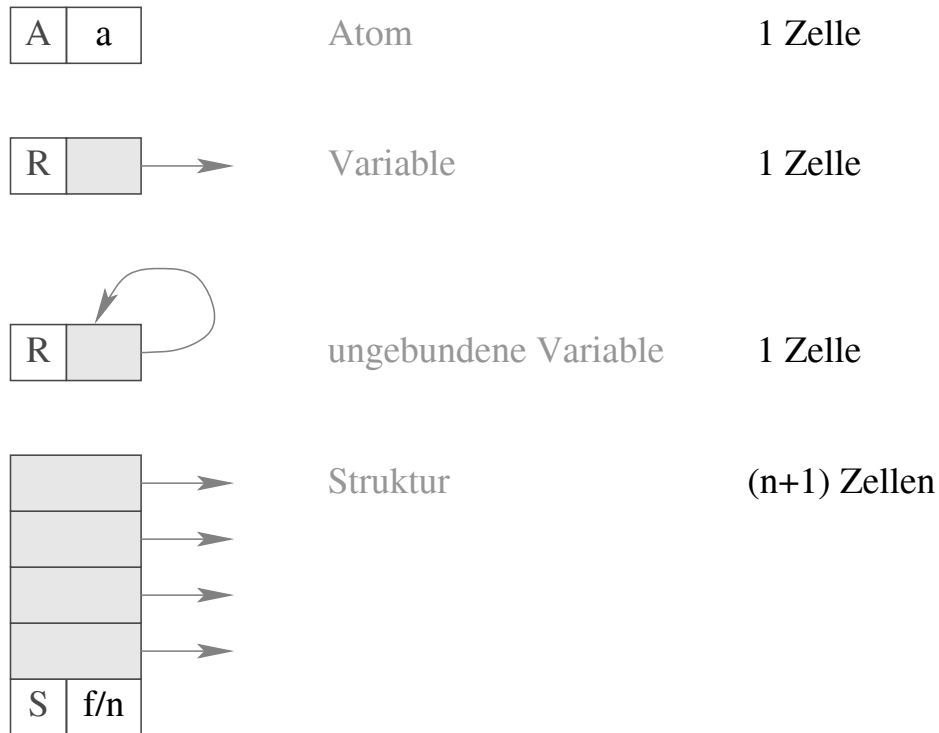


Abb. 4.3. Die Haldenobjekte der WiM.

Referenzen oder Variablen. Eine *ungebundene* Variable repräsentieren wir dabei als eine *Selbstreferenz*. Dies gestattet einen einfachen Test, ob eine Variable gebunden ist, d.h. bereits auf einen anderen Term verweist oder nicht.

4.3 Anlegen von Termen in der Halde

Argumente von Zielen (Aufrufen) werden vor der Übergabe in der Halde aufgebaut. Nehmen wir an, wir hätten eine Adressumgebung ρ , die für jede Variable X die Adresse (relativ zu FP) auf dem Keller liefert.

Dann wollen wir für einen Term t eine Folge $\text{code}_A t \rho$ von Instruktionen erzeugen, die eine Repräsentation von t in der Halde aufbaut und eine Referenz darauf auf dem Keller zurückliefert.

Wie könnte der Code dafür aussehen? Die einfachste Idee besteht darin, den Term in Postorder zu durchlaufen. Bei der Abarbeitung eines Knotens liegen dann die Referenzen auf die Nachfolger bereits oben auf dem Keller, genau richtig, um das entsprechende Haldenobjekt für den Knoten aufzubauen. Für jede mögliche Art von Knoten benötigen wir deshalb eine geeignete WiM-Instruktion.

Beispiel 4.3.1 Betrachten wir den folgenden Term:

$$f(g(X, Y), a, Z)$$

Dabei soll X bereits initialisiert sein, d.h. $S[FP + \rho(X)]$ enthält bereits eine Referenz. Die Variablen Y und Z dagegen sollen noch ungebunden sein. Dann muss die Halde erweitert werden wie in Abb. 4.4. Beachten Sie, dass unsere Allokationsstra-

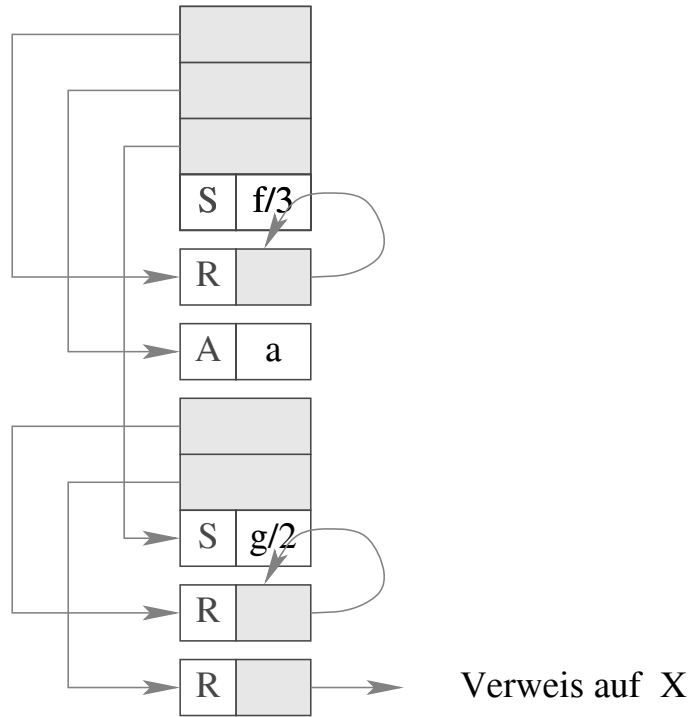


Abb. 4.4. Die Halde für den Term $f(g(\bar{X}, Y), a, Z)$.

ategie dazu führt, dass stets Verweise in Haldenzellen mit größeren Adressen auf Haldenadressen mit kleineren (oder gleichen) Adressen zeigen. Diese Verweisrichtung werden wir nach Möglichkeit stets zu erzielen suchen. Sie bedeutet nämlich, dass bei Freigabe jüngerer Haldenobjekte keine hängenden Verweise auf bereits freigegebene Objekte (Dangling References) entstehen. \square

In unserem Beispiel 4.3.1 wird deutlich, dass gebundene und ungebundene Variablen unterschiedlich behandelt werden müssen. Zur Unterscheidung kennzeichnen wir deshalb Vorkommen bereits initialisierter Variablen mit einem Oberstrich (im Beispiel \bar{X}). Beachten Sie, dass Argumente immer initialisiert sind, anonyme Variablen dagegen nie! Gemäß dieser Diskussion definieren wir:

```
codeA a ρ = putatom a
codeA X ρ = putvar ρ(X)
codeA  $\bar{X}$  ρ = putref ρ(X)
codeA _ ρ = putanon
```

$$\begin{aligned}
\text{code}_A f(t_1, \dots, t_n) \rho &= \text{code}_A t_1 \rho \\
&\dots \\
&\text{code}_A t_n \rho \\
&\mathbf{putstruct} f/n
\end{aligned}$$

Beispiel 4.3.2 Für den Term $f(g(\bar{X}, Y), a, Z)$ aus Beispiel 4.3.1 bei einer Adressumgebung $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ ergibt das die Folge:

putref 1	putatom a
putvar 2	putvar 3
putstruct $g/2$	putstruct $f/3$

□

Offenbar können wir so sehr leicht Code zum Aufbauen von Termen generieren. Wir werden jetzt die dabei verwendeten Instruktionen definieren.

Die Instruktion **putatom** a legt ein Atom auf der Halde an (Abb. 4.5).

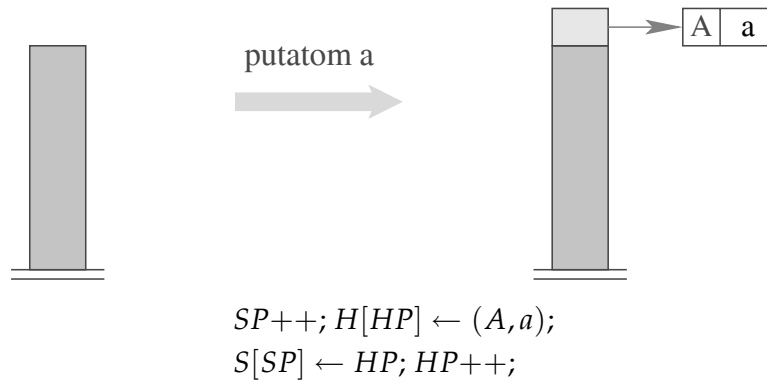


Abb. 4.5. Die Instruktion **putatom** a .

Die Instruktion **putvar** i erzeugt eine neue uninitialisierte Variable und initialisiert zusätzlich die entsprechende Zelle im Kellerrahmen (Abb. 4.6).

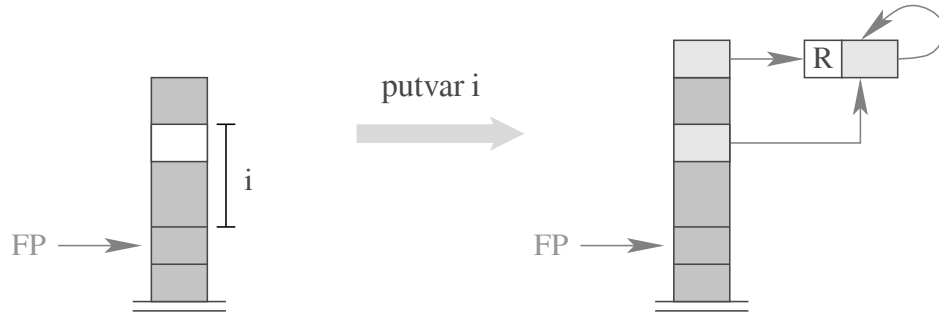
Die Instruktion **putanon** führt eine neue ungebundene Variable ein, legt die Referenz darauf oben auf den Keller, speichert aber keine Referenz darauf im Kellerrahmen (Abb. 4.7). Die Instruktion **putref** i kopiert nicht einfach den Verweis aus $S[FP + i]$ oben auf den Keller, sondern dereferenziert ihn vorher so oft wie möglich. Dazu verwenden wir die Hilfsfunktion *deref*():

```

ref deref (ref  $v$ ) {
    if ( $H[v] = (R, w) \wedge v \neq w$ ) return deref ( $w$ );
    else return  $v$ ;
}

```

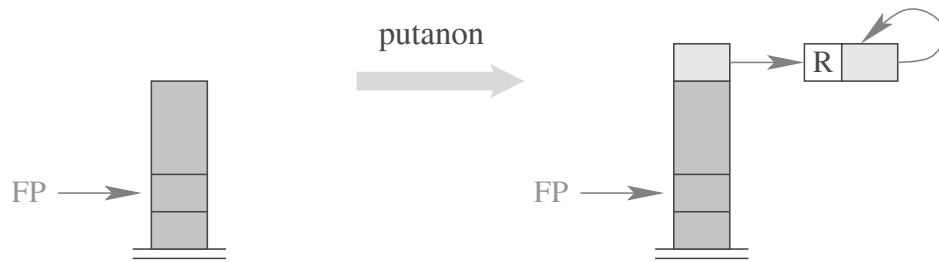
Die Instruktion **putstruct** f/n erzeugt eine Anwendung des Konstruktors f/n in der Halde (Abb. 4.9). Unser Übersetzungsschema zusammen mit den Definitionen der



$$SP++; H[HP] \leftarrow (R, HP);$$

$$S[FP + i] \leftarrow S[SP] \leftarrow HP; HP++;$$

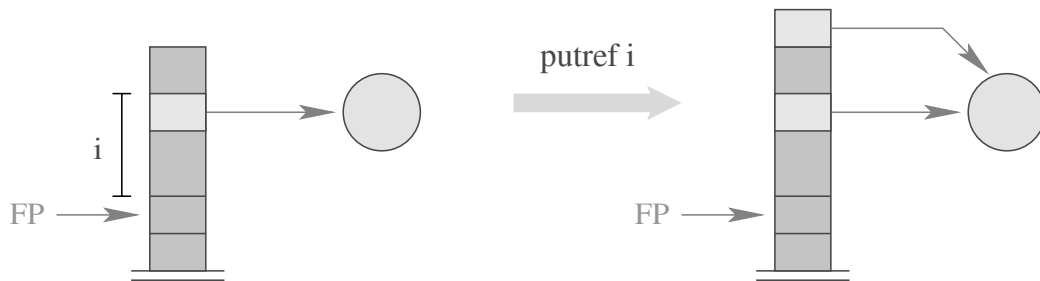
Abb. 4.6. Die Instruktion **putvar i** .



$$SP++; H[HP] \leftarrow (R, HP);$$

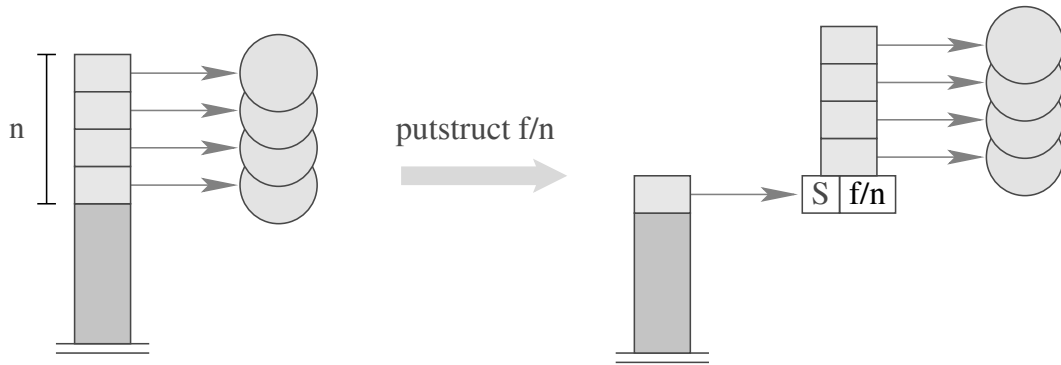
$$S[SP] \leftarrow HP; HP++;$$

Abb. 4.7. Die Instruktion **putanon**.



$$SP++; S[SP] \leftarrow \text{deref}(S[FP + i]);$$

Abb. 4.8. Die Instruktion **putref i** .



$$\begin{aligned}
 &SP \leftarrow SP - n + 1; H[HP] \leftarrow (S, f/n); \\
 &\text{for } (i \leftarrow 0; i < n; i++) H[HP + 1 + i] \leftarrow S[SP + i]; \\
 &S[SP] \leftarrow HP; HP \leftarrow HP + n;
 \end{aligned}$$

Abb. 4.9. Die Instruktion **putstruct** f/n .

neuen virtuellen Instruktionen sollte Sie davon überzeugt haben, dass nicht nur für den Beispielterm aus Beispiel 4.3.1, sondern allgemein in den aufgebauten Termen die Referenzen stets auf kleinere Haldenadressen zeigen.

4.4 Die Übersetzung von Literalen

Wenden wir uns der Übersetzung von Literalen zu. Literale in ProL entsprechen Prozeduraufrufen in imperativen Sprachen. Entsprechend legen wir zu ihrer Auswertung einen Kellerrahmen an. Dann konstruieren wir Repräsentationen der Terme, welche als aktuelle Parameter fungieren, in der Halde und speichern Verweise darauf im Kellerrahmen. Schließlich springen wir den Code für die Prozedur bzw. das Prädikat an. Das ergibt das folgende Übersetzungsschema für Literale:

$$\begin{aligned}
 \text{code}_G q(t_1, \dots, t_k) \rho = & \quad \text{mark } B && // \text{legt einen Kellerrahmen an} \\
 & \text{code}_A t_1 \rho \\
 & \dots \\
 & \text{code}_A t_k \rho \\
 & \text{call } q/k && // \text{ruft Prozedur } q/k \text{ auf} \\
 B : & \dots
 \end{aligned}$$

Beispiel 4.4.1 Betrachten Sie das Literal $q(a, X, g(\bar{X}, Y))$. In einer Adressumgebung $\rho = \{X \mapsto 1, Y \mapsto 2\}$ ergibt unser Codeerzeugungsschema:

mark B	putref 1	call $q/3$
putatom a	putvar 2	$B :$...
putvar 1	putstruct $g/2$	

□

Bevor wir die WiM-Instruktion **mark** B implementieren können, müssen wir uns darüber klar werden, wie ein Kellerrahmen der WiM aufgebaut ist (Abb. 4.10). Die einfachst mögliche Organisation besteht darin, zuerst die organisatorischen Zellen innerhalb des Rahmens anzulegen. Darüber legen wir die aktuellen Parameter des Literals, möglicherweise gefolgt von weiteren lokalen Variablen einer gerade ausgewählten Klausel. Wie bei der virtuellen Maschine für C können wir auch bei der Übersetzung logischer Programmiersprachen die Parameter und lokalen Variablen relativ zum FP adressieren. Der jeweils aktuelle FP zeigt dabei wie üblich auf die letzte organisatorische Zelle. An organisatorischen Zellen kennen wir bereits zwei: Eine Zelle benötigen wir zur Rettung des aktuellen Standes des PC , der *positiven Fortsetzungsadresse*, d.h. der Stelle im Code, an der bei erfolgreicher Abarbeitung des Literals fortgefahren werden soll. Dies entspricht der Rücksprungadresse etwa bei der C-Maschine. Eine weitere Zelle benötigen wir, um den Inhalt des Registers FP vor dem Aufruf zu retten. Es wird sich zeigen, dass wir zur Implementierung des Backtracking vier weitere Register und deshalb auch ebenso viele zusätzliche organisatorische Zellen benötigen.

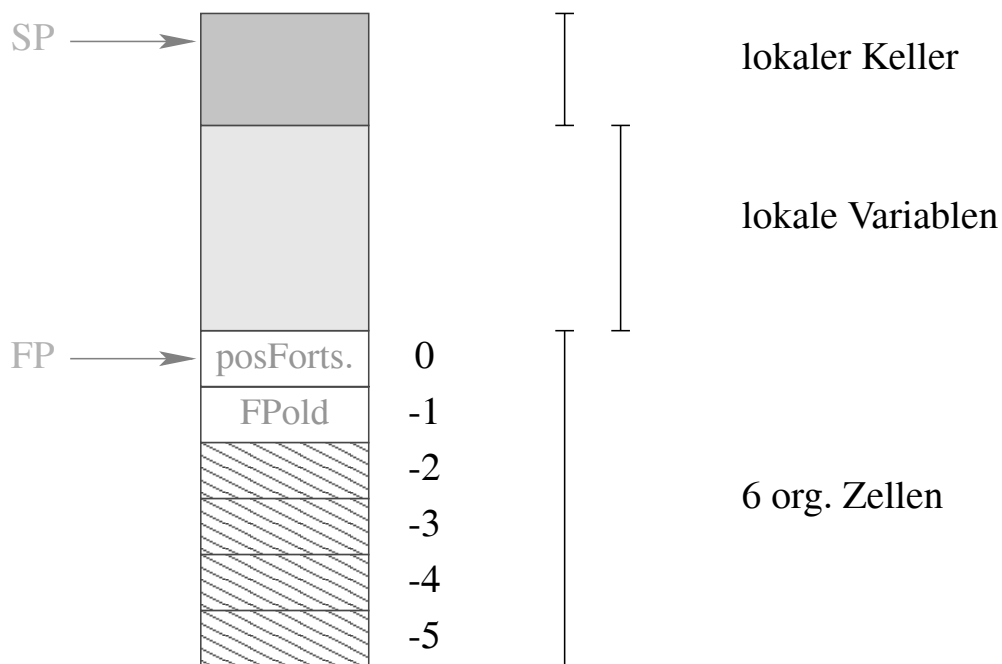
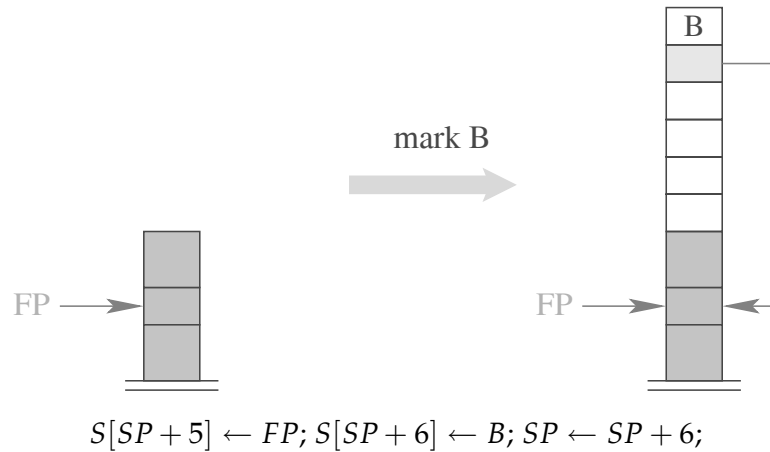
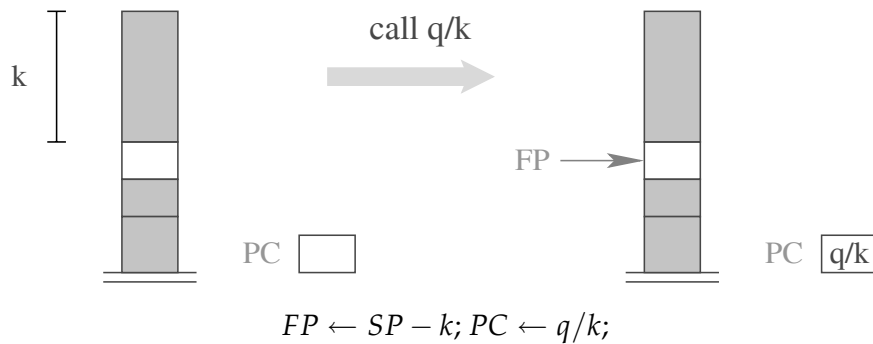


Abb. 4.10. der Kellerrahmen der WiM.

Damit haben wir alle notwendigen Informationen, um die Instruktion **mark** B zu implementieren (Abb. 4.11). Diese Instruktion allokiert auf dem Keller Platz für sechs organisatorische Zellen. Dann rettet sie den aktuellen FP und speichert B als positive Fortsetzungsadresse ab.

Die Instruktion **call** q/k setzt das Register FP auf die oberste organisatorische Zelle des Kellerrahmens und ruft das k -stellige Prädikat q/k auf (Abb. 4.12).

**Abb. 4.11.** Die Instruktion **mark B**.**Abb. 4.12.** Die Instruktion **call q/k**.

4.5 Unifikation

Wir kommen zur wichtigsten Operation bei der Ausführung eines logischen Programms, der *Unifikation*. Ziel der Unifikation zweier Terme ist es, eine minimale oder *allgemeinste* Substitution der in den Termen vorkommenden Variablen zu finden, so dass die beiden Terme gleich werden.

Beispiel 4.5.1 Betrachten Sie die beiden Terme:

$$f(X, g(a, X)) \quad \text{und} \quad f(h(b), g(Y, Z))$$

Eine Substitution, die die beiden Terme unifiziert, ist gegeben durch:

$$\{X \mapsto h(b), Y \mapsto a, Z \mapsto h(b)\}$$

In diesem Beispiel gibt es nur diese eine unifizierende Substitution. Das ist jedoch nicht immer der Fall. Für die beiden Terme:

$$f(X, g(a, X)) \quad \text{und} \quad f(h(Z), g(a, Y))$$

gibt es unendlich viele solche Substitutionen. Eine *minimale* unifizierende Substitution, d.h. eine, die so wenige Variablen wie möglich ersetzt, für diese beiden Terme ist gegeben durch:

$$\{X \mapsto h(Z), Y \mapsto h(Z)\}$$

Eine solche minimale unifizierende Substitution hat die schöne Eigenschaft, dass alle anderen unifizierenden Substitutionen aus ihr gewonnen werden können, indem man weitere Variablen ersetzt. Deshalb nennen wir eine solche unifizierende Substitution auch *allgemeinste* (Most General Unifier).

Wie Sie bereits vermutet haben, kann eine Unifikation auch *fehlschlagen*. Betrachten Sie z.B. die beiden Terme:

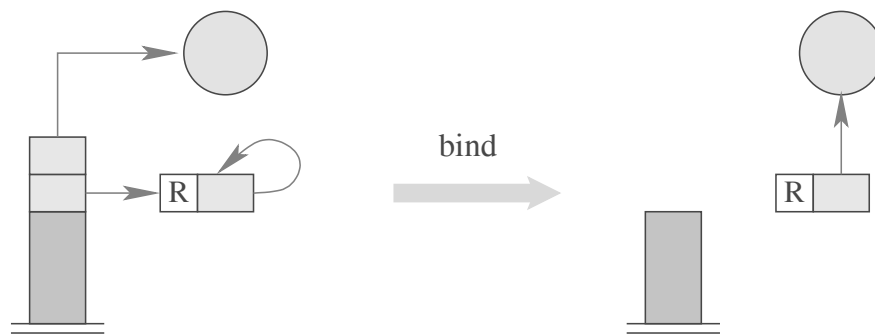
$$f(X, a) \quad \text{und} \quad f(h(Z), g(a, Y))$$

Dann gibt es keine Substitution der Variablen, die die beiden Terme gleich macht. Ein spezieller Fall liegt vor, wenn eine Variable X mit einem Term unifiziert werden soll, der nicht selbst gleich X ist, aber die Variable X enthält, wie z.B. $f(X, a)$. Offenbar kann es in dieser Situation ebenfalls keine unifizierende Substitution geben – zumindest keine, die Variablen an *endliche* Terme bindet. \square

Wir wollen Unifikationsgleichungen $X = t$ übersetzen. Ist die Variable X auf der linken Seite uninitialisiert, setzen wir:

$$\begin{aligned} \text{code}_G (X = t) \rho &= \text{putvar } \rho(X) \\ &\quad \text{code}_A t \rho \\ &\quad \text{bind} \end{aligned}$$

Die Implementierung muss in diesem Fall den Term t aufbauen und das Referenzobjekt von X an t binden. Diese letzte Aufgabe leistet die Instruktion **bind** (Abb. 4.13.). Beachten Sie, dass die Variable, die durch die Instruktion **bind** gebunden



$$H[S[SP - 1]] \leftarrow (R, S[SP]); \text{trail}(S[SP - 1]); SP \leftarrow SP - 2;$$

Abb. 4.13. Die Instruktion **bind**.

wird, gegebenenfalls durch den Aufruf der Funktion *trail* mitprotokolliert wird.

Damit müssen wir nur noch die Unifikation für eine bereits gebundene Variable X realisieren. Die erste Idee besteht darin, wie im Falle einer ungebundenen Variable erst eine Referenz auf (die Bindung von) X zu kellern und darüber eine Referenz auf den Term t . Die Instruktion **unify** soll dann die Unifikation der beiden Terme implementieren, d.h. die notwendigen Bindungen der in den beiden Termen vorkommenden Variablen vornehmen. Damit ergibt sich als Übersetzungsschema:

$$\begin{aligned} \text{code}_G (\bar{X} = t) \rho = & \text{putref } \rho(X) \\ & \text{code}_A t \rho \\ & \text{unify} \end{aligned}$$

Beispiel 4.5.2 Betrachten Sie die Gleichung:

$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Für die Adressumgebung:

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

erhalten wir:

putref 4	putref 1	putatom a	unify
	putvar 2	putvar 3	
	putstruct g/2	putstruct f/3	

□

Die Instruktion **unify** ruft die Hilfsfunktion *unify()* für die beiden obersten Kellerreferenzen auf (Abb. 4.14).

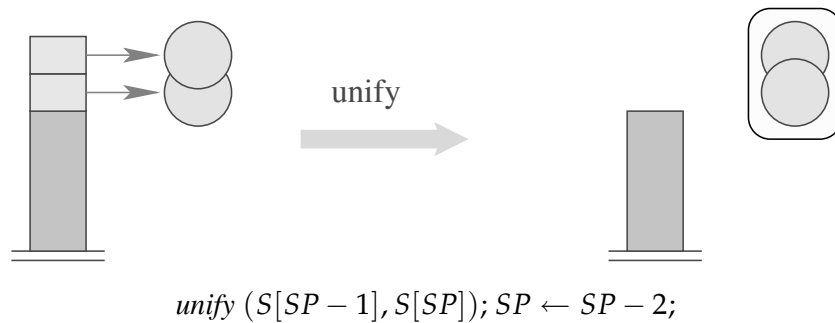


Abb. 4.14. Die Instruktion **unify**.

Die Funktion *unify()* des Laufzeitsystems erhält als Eingabe zwei Haldenadressen und führt die Unifikation durch. Dabei beachtet sie, dass gleiche Haldenadressen bereits unifiziert sind. Beim Binden zweier Variablen aneinander sollte stets die jüngere mit der größeren Adresse an die ältere mit der kleineren Adresse gebunden werden. Beim Binden einer Variablen an einen Term muss überprüft werden, ob diese

Variable nicht in diesem Term vorkommt. Diese Überprüfung nennt man auch *Vorkommenstest* (Occur-Check). Weiterhin müssen eingegangene Bindungen mitprotokolliert werden, damit sie später gegebenenfalls wieder rückgängig gemacht werden können. Schließlich kann die Unifikation auch fehlschlagen. In diesem Fall muss zurückgesetzt werden (Backtracking).

```

bool unify (ref u, ref v) {
    if (u = v) return true;
    if (H[u] = (R, _)) {
        if (H[v] = (R, _)) {
            if (u > v) {H[u] ← (R, v); trail (u); return true;}
            else {H[v] ← (R, u); trail (v); return true;}
        } else if (check (u, v)) {
            H[u] ← (R, v); trail (u); return true;
        } else {backtrack(); return false;}
    }
    if ((H[v] = (R, _)) {
        if (check (v, u)) {
            H[v] ← (R, u); trail (v); return true;
        } else {backtrack(); return false;}
    }
    if (H[u] = (A, a) ∧ H[v] = (A, a)) return true;
    if (H[u] = (S, f/n) ∧ H[v] = (S, f/n)) {
        for (int i ← 1; i ≤ n; i++)
            if (¬unify (deref (H[u + i]), deref (H[v + i])) return false;
        return true;
    }
    backtrack(); return false;
}

```

Die Unifikation der Terme in Abb. 4.15 setzt also Referenzen entsprechend Abb. 4.16. Beachten Sie, dass unsere Implementierung, wann immer möglich, Referenzen an größeren Adressen auf Objekte mit kleineren Adressen zeigen lässt.

Unsere Implementierung verwendet drei Hilfsfunktionen. Die Hilfsfunktion *trail()* protokolliert neue Bindungen. Die Hilfsfunktion *backtrack()* initiiert das Rücksetzen. Diese beiden Funktionen diskutieren wir später. Die Hilfsfunktion *check()* implementiert den Vorkommenstest. Sie überprüft, ob die Variable im ersten Argument in dem Term vorkommt, auf dessen Repräsentation das zweite Argument zeigt. Aus Effizienzgründen wird dieser Test jedoch in manchen Implementierungen weggelassen. Dann gilt einfach:

```

bool check (ref u, ref v) { return true; }

```

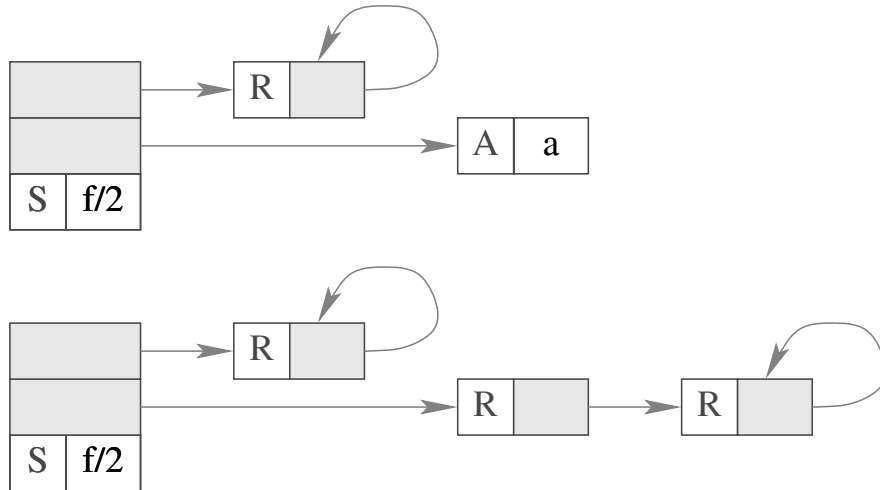


Abb. 4.15. Eine Beispielleingabe für die Funktion *unify*().

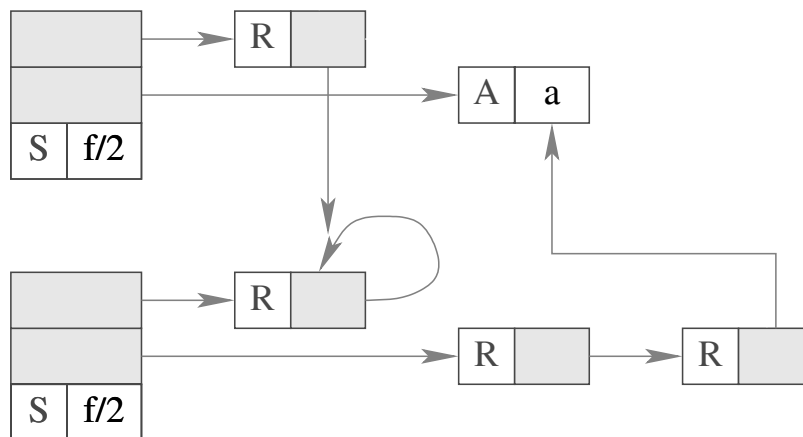


Abb. 4.16. Das Ergebnis des Aufrufs von *unify*() aus Abb. 4.15.

Ansonsten könnte man ihn wie folgt implementieren:

```

bool check (ref u, ref v) {
    if (u = v) return false;
    if (H[v] = (S, f/n))
        for (int i ← 1; i ≤ n; i++)
            if (¬check (u, deref (H[v + i]))) return false;
    return true;
}

```

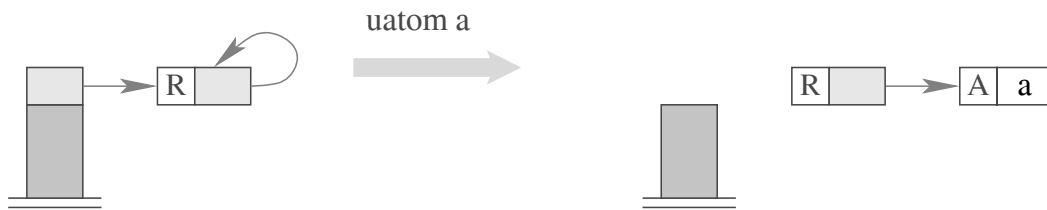
Unsere bisherige Übersetzung der Gleichung $\bar{X} = t$ ist sehr einfach. Ihr Nachteil ist jedoch, dass die gerade konstruierten Haldenobjekte oft sofort Müll werden. Eine bessere Implementierung kann das überflüssige Anlegen solcher Objekte vermeiden. Dazu wird ein Verweis auf die aktuelle Bindung von X oben auf den Keller gelegt mit dem Ziel, den Term t auf der rechten Seite der Unifikationsgleichung in eine Instruktionsfolge zu übersetzen, die die Unifikation mit t implementiert. Wir definieren deshalb:

$$\text{code}_G (\bar{X} = t) \rho = \text{putref } \rho(X) \\ \text{code}_U t \rho$$

für eine neue Übersetzungsfunktion code_U . Betrachten wir zuerst den Code für Atome und Variablen:

$$\begin{aligned} \text{code}_U a \rho &= \text{uatom } a \\ \text{code}_U X \rho &= \text{uvar } \rho(X) \\ \text{code}_U _ \rho &= \text{pop} \\ \text{code}_U \bar{X} \rho &= \text{uref } \rho(X) \\ &\dots \text{ // wird fortgesetzt.} \end{aligned}$$

Die Instruktion **uatom** a implementiert die Unifikation mit dem Atom a (Abb. 4.17). Der Aufruf von *trail()* vermerkt dabei die vorgenommene Bindung. Den Auf-



```

h ← S[SP]; SP--;
switch (H[h]) {
  case (A, a) : break;
  case (R, _) : H[HP] ← (A, a);
                H[h] ← (R, HP); trail (h);
                HP++; break;
  default :    backtrack();
}

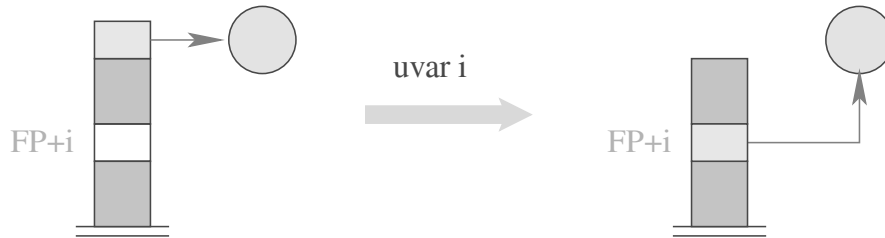
```

Abb. 4.17. die Instruktion **uatom** a .

ruf von *backtrack()* benötigen wir, um im Falle eines Fehlschlags der Unifikation die Rücksetzung zu initiieren.

Da eine Unifikation mit einer anonymen Variable keine Variablen bindet, können wir diese durch die Instruktion **pop** implementieren.

Die Instruktion **uvar** i implementiert die Unifikation mit der i -ten Variable, sofern diese ungebunden ist (Abb. 4.18). Diese Unifikation bindet die i -te lokale Variable und schlägt somit nie fehl. Die Instruktion **uref** i implementiert die Unifikation



$$S[FP + i] \leftarrow S[SP]; SP--;$$

Abb. 4.18. Die Instruktion **uvar** i .

mit der i -ten Variable, sofern diese gebunden ist (Abb. 4.19). Beachten Sie, dass der Verweis für die i -te Variable vor der Unifikation maximal dereferenziert wird. Da in diesem Fall zur Übersetzungszeit keine Informationen über die zu unifizierenden Terme vorliegen, müssen wir explizit die Laufzeitfunktion *unify*() aufrufen.



$$\text{unify}(S[SP], S[FP + i]); SP--;$$

Abb. 4.19. Die Instruktion **uref** i .

Damit haben wir alle Möglichkeiten für einfache Terme behandelt. Für zusammengesetzte Terme t muss der Code für die Unifikation einen *pre-order* Durchlauf über den Term t durchführen. Soll der aktuelle Term t dabei mit einer ungebundenen Variable unifiziert werden, wird vom Vergleich auf den Aufbau des Terms umgeschaltet. Insgesamt definieren wir damit:

$$\begin{aligned}
\text{code}_U f(t_1, \dots, t_n) \rho = & \quad \mathbf{ustruct} \ f/n \ A \\
& \mathbf{son} \ 1 \\
& \text{code}_U t_1 \rho \\
& \dots \\
& \mathbf{son} \ n \\
& \text{code}_U t_n \rho \\
& \mathbf{up} \ B \\
A : & \text{check} \ (\text{ivars}(f(t_1, \dots, t_n))) \ \rho \\
& \text{code}_A f(t_1, \dots, t_n) \ \rho \\
& \mathbf{bind} \\
B : & \dots
\end{aligned}$$

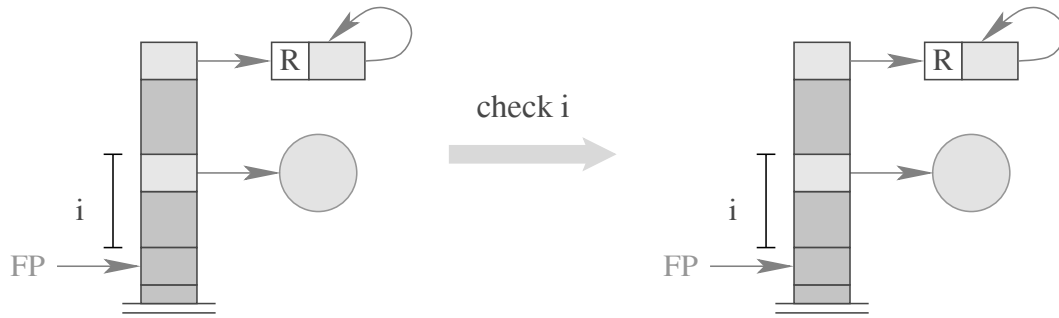
Wir beginnen die Diskussion dieses Übersetzungsschemas mit dem Abschnitt zwischen den symbolischen Marken A und B . Diesen Abschnitt nennen wir den *Aufbaublock* für den Term t . Ihn springt die Instruktion **ustruct** $f/n \ A$ an, falls die Referenz oben auf dem Keller, mit der (die Repräsentation von) t unifiziert werden soll, eine ungebundene Variable X' darstellt. Ist das der Fall, muss weiter überprüft werden, ob X' in der aktuellen Instanz von t vorkommt. Falls dies nicht der Fall ist, wird eine Repräsentation von t materialisiert, um anschließend die Referenz für X' auf diese Repräsentation umzusetzen.

Die Variable X' kommt in der aktuellen Instanz von t vor, genau dann wenn X' in der Bindung einer der bereits initialisierten Variablen vorkommt, die in t enthalten ist. In unserer Übersetzungsschema bezeichnet dabei $\text{ivars}(t)$ die Menge der bereits initialisierten Variablen des Terms t . Die für eine in $\text{ivars}(t)$ enthaltene Variable Y_j nötige Überprüfung geschieht durch die Instruktion **check** $\rho(Y_j)$:

$$\begin{aligned}
\text{check} \ \{Y_1, \dots, Y_d\} \ \rho = & \quad \mathbf{check} \ \rho(Y_1) \\
& \mathbf{check} \ \rho(Y_2) \\
& \dots \\
& \mathbf{check} \ \rho(Y_d)
\end{aligned}$$

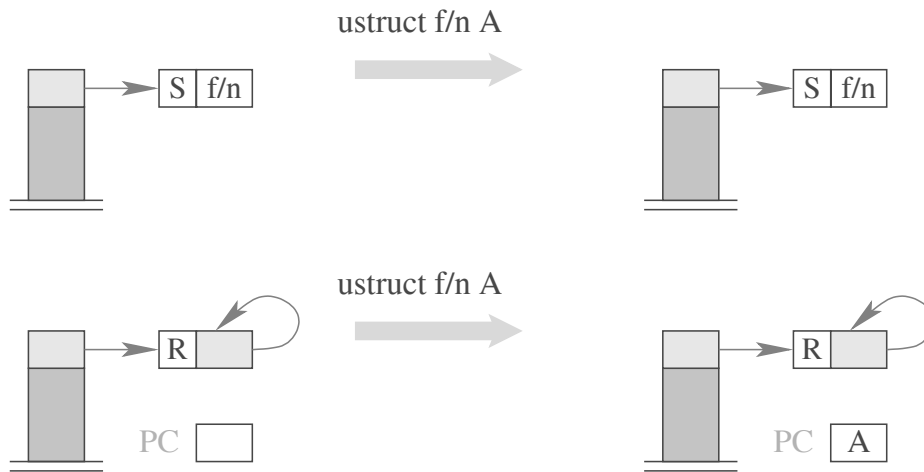
Die Instruktion **check** i überprüft, ob die (ungebundene) Variable oben auf dem Keller innerhalb des Terms vorkommt, an den die Variable mit Relativadresse i gebunden ist (Abb. 4.20). Ist dies der Fall, schlägt die Unifikation fehl und Rücksetzen wird veranlasst. Die Instruktion **bind** schließt den Term-Aufbau ab. Sie bindet die (ungebundene) Variable oben auf dem Keller an den konstruierten Term.

Wenden wir uns nun dem Code zur Implementierung der Unifikation zu. Die zugehörige Instruktionsfolge implementiert einen *Preorder*-Durchlauf durch den Term t . Dieser Durchlauf beginnt an der Wurzel von t mit der Instruktion **ustruct** $f/n \ A$ (Abb. 4.21). Beachten Sie, dass die Instruktion **ustruct** $f/n \ A$ die Argumentreferenz auf dem Keller belässt. Sie testet, ob die oberste Referenz auf eine ungebundene Variable oder eine Struktur f/n zeigt. Ist beides nicht der Fall, wird Rück-



```
if ( $\neg$ check ( $S[SP]$ , deref ( $S[FP + i]$ ))) backtrack();
```

Abb. 4.20. Die Instruktion **check i** .

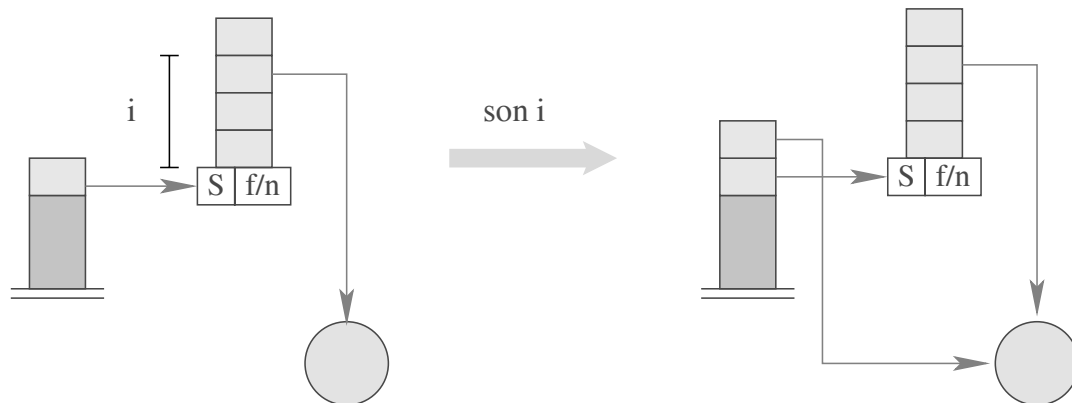


```
switch (H[S[SP]]) {  
  case (S, f/n) : break;  
  case (R, _) : PC  $\leftarrow$  A; break;  
  default : + backtrack();
```

Abb. 4.21. Die Instruktion **ustruct f/n A**.

setzen ausgelöst. Zeigt die oberste Referenz auf eine (ungebundene) Variable, wird der Aufbaublock ab Adresse A angesprungen.

Zeigt die oberste Referenz dagegen auf eine Struktur f/n , steigen man rekursiv zu den Kindern ab. Anschließend wird der Keller bereinigt und hinter dem Code an der Adresse B fortgefahren. Für den Abstieg zum i -ten Kind führen wir die Instruktion **son** i ein (Abb. 4.22). Sie legt die i -te Referenz der Struktur, auf die die oberste

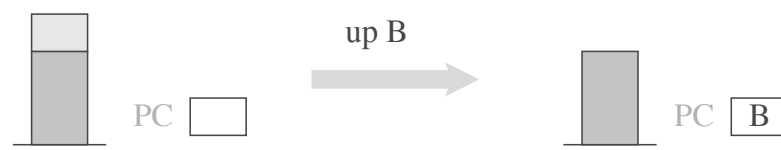


$$S[SP + 1] \leftarrow \text{deref}(H[S[SP] + i]); SP++;$$

Abb. 4.22. Die Instruktion **son** i .

Referenz zeigt, oben auf den Keller.

Der Bereinigung des Kellers nach Überprüfung sämtlicher Teilbäume von t dient die Instruktion **up** B (Abb. 4.23). Diese Instruktion wirft die oberste Referenz vom Keller und setzt anschließend den PC auf die Adresse B . In unserer Anwendung ist



$$SP--; PC \leftarrow B;$$

Abb. 4.23. Die Instruktion **up** B .

diese Adresse B die positive Fortsetzungsadresse, d.h. die erste Adresse hinter dem Aufbaublock.

Beispiel 4.5.3 Für den Term $f(g(\bar{X}, Y), a, Z)$ und die Adressumgebung $\{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ erhalten wir:

ustruct $f/3$ A_1	up B_2	B_2 :	son 2	putvar 2
son 1			uatom a	putstruct $g/2$
ustruct $g/2$ A_2	A_2 :	check 1	son 3	putatom a
son 1		putref 1	uvar 3	putvar 3
uref 1		putvar 2	up B_1	putstruct $f/3$
son 2		putstruct $g/2$ A_1 :	check 1	bind
uvar 2		bind	putref 1 B_1 :	...

□

Bei der Implementierung der Unifikations-Instruktionen haben wir genau darauf geachtet, dass die oben auf den Keller gelegten Referenzen stets maximal dereferenziert sind.

Beachten Sie schließlich, dass wir hier gegen das oberste Gebot der Codeerzeugung verstoßen haben, für jedes Programmstück nur eine Instruktionsfolge zu erzeugen. Tatsächlich wird für tief geschachtelte Terme t für den selben Teilterm mehrmals Aufbau-code erzeugt. Für tiefe Terme kann deshalb die Codegröße beträchtlich ansteigen. Dieses Problem ist jedoch insofern vernachlässigbar, als tiefe Terme bei der Programmierung in Prolog eher selten auftreten.

4.6 Klauseln

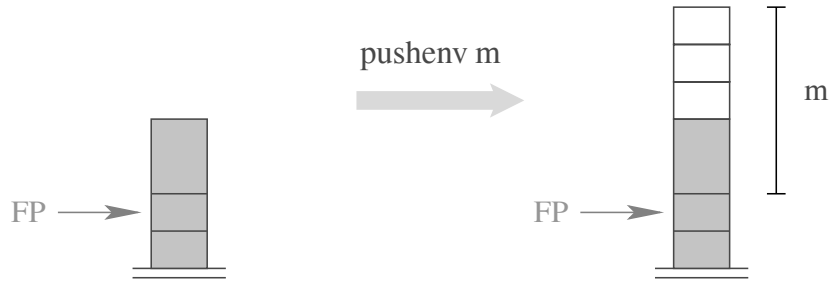
Wenden wir uns nun der Codeerzeugung für Klauseln zu. Der Code für eine Klausel muss zuerst Platz für die lokalen Variablen der Klausel allokieren. Anschließend benötigen wir Code zur Auswertung des Rumpfs. Schließlich sollte der Kellerrahmen aufgegeben werden – zumindest immer, wenn das möglich ist.

Sei r die Klausel $q(X_1, \dots, X_k) \Leftarrow g_1, \dots, g_n$. Sei weiterhin $\{X_1, \dots, X_m\}$, $m \geq k$, die Menge der lokalen Variablen von r . Erinnern Sie sich, dass wir vereinbart haben, diese Variablen ab Relativadresse 1 im Kellerrahmen abzulegen. Sei dementsprechend ρ die Adressumgebung mit $\rho(X_i) = i$, für $i = 1, \dots, m$. Dann übersetzen wir:

```
codeC  $r$  = pushenv  $m$            //reserviert Platz für lokale Variablen
          codeG  $g_1$   $\rho$ 
          ...
          codeG  $g_n$   $\rho$ 
          popenv
```

Der neue Befehl **popenv** restauriert FP und PC und versucht, den aktuellen Kellerrahmen freizugeben. Das sollte gelingen, wenn die Programmausführung nie mehr zu diesem Kellerrahmen zurückkehrt. Genauer werden wir den Befehl **popenv** in Kapitel 4.8 diskutieren.

Zur Allokation von m lokalen Variablen setzt der Befehl **pushenv** m den SP auf $FP + m$ (Abb. 4.24).



$$SP \leftarrow FP + m;$$

Abb. 4.24. Die Instruktion **pushenv** m .

Beispiel 4.6.1 Betrachten Sie die Klausel r :

$$a(X, Y) \Leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Dann liefert $\text{code}_C r$ in der Adressumgebung $\{X \mapsto 1, Y \mapsto 2, X_1 \mapsto 3\}$:

pushenv 3	mark A	$A :$	mark B	$B :$	popenv
	putref 1		putref 3		
	putvar 3		putref 2		
	call $f/2$		call $a/2$		

□

4.7 Die Übersetzung von Prädikaten

Jedes Prädikat q/k wird definiert durch eine Folge von Klauseln $rr \equiv r_1 \dots r_f$.

Die Übersetzung des Prädikats q/k enthält sicherlich die Übersetzungen der einzelnen Klauseln r_i . Insbesondere haben wir für $f = 1$:

$$\text{code}_P rr = \text{code}_C r_1$$

Falls q/k jedoch durch mehrere Klauseln definiert ist, muss die erste Alternative angesprungen werden. Schlägt diese fehl, werden der Reihe nach die weiteren Alternativen probiert. Bei der Codeerzeugung für Prädikate reicht also eine Hintereinandersetzung der Übersetzungen der einzelnen Klauseln nicht aus. Vielmehr benötigen wir eine Implementierung, die einzelne Alternativen versuchsweise ausführt und bei

Fehlschlag ihren gesamten Effekt wieder rückgängig macht. Im nächsten Abschnitt betrachten wir darum nun das Rücksetzen von Berechnungen (Backtracking) etwas näher.

4.7.1 Backtracking

Wir haben schon gesehen, dass, wenn die Unifikation fehlschlägt, die Laufzeitfunktion *backtrack()* aufgerufen wird. Durch diesen Aufruf soll die gesamte Berechnung bis zum *dynamisch* letzten Ziel rückgängig gemacht werden, an dem eine alternative Klausel gewählt werden kann. Den zugehörigen Kellerrahmen auf dem Laufzeitkeller nennen wir den aktuellen *Rücksetzpunkt* (Backtrack Point). Um zwischenzeitlich eingegangene Variablenbindungen aufzuheben, werden die eingegangenen neuen Bindungen mithilfe der Laufzeitfunktion *trail()* mitprotokolliert. Die Aufrufe der Funktion *trail()* speichern kritische Variablen in einer speziellen Datenstruktur, der *Spur* (Trail, Abb. 4.25). Die Spur ist ein weiterer Keller *T*, auf dem

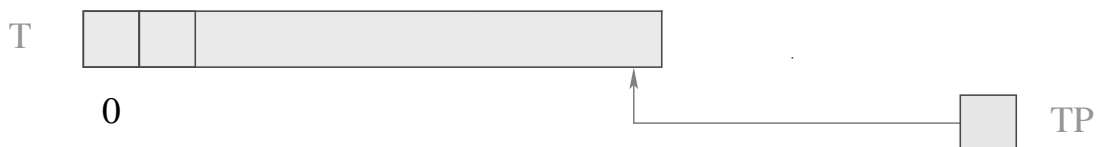


Abb. 4.25. Die WiM-Datenstruktur *Trail*.

Haldenadressen von Referenzobjekten gemerkt werden, die eine neue Bindung erhalten. Eine Optimierung würde sich dabei nur Adressen *a* merken, deren Referenzobjekte (R, b) Bindungen an zu junge Haldenobjekte enthalten. Zur Verwaltung der Spur führen wir ein weiteres Register ein, den *Spurzeiger* (Trail Pointer) *TP*. Der Spurzeiger zeigt jeweils auf die oberste belegte Spurzelle.

Weiterhin benötigen wir ein neues Register *BP*, den *Rücksetzzeiger* (Backtrack Pointer). Er zeigt auf den aktuellen Rücksetzpunkt, d.h. den letzten Kellerrahmen, in dem noch eine weitere Alternative offen ist (Abb. 4.26).

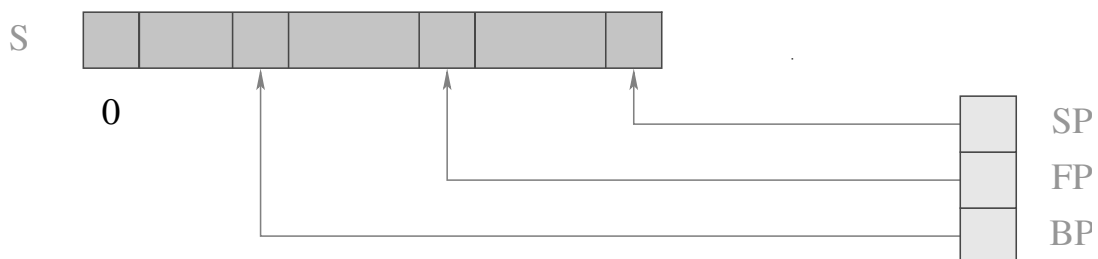


Abb. 4.26. Das Register *BP* der WiM.

In jedem Kellerrahmen haben wir bisher den alten Wert des *PC*, d.h. die *positive* Fortsetzungsadresse, sowie den alten Wert des *FP* abgelegt. Dieser ist der Verweis auf den aufrufenden Kellerrahmen, d.h. den dynamischen Vorgänger (Abb. 4.27). Im Kellerrahmen eines Rücksetzpunkts benötigen wir zusätzlich die Codeadresse für die *nächste* Alternative, d.h. die *negative Fortsetzungsadresse* sowie den alten Wert des *BP*. Den beiden dafür notwendigen organisatorischen Zellen geben wir die Relativadressen -5 und -4 . Weiterhin retten wir an einem Rücksetzpunkt ebenfalls die alten Werte der Register *TP* und *HP*. Die zugehörigen organisatorischen Zellen

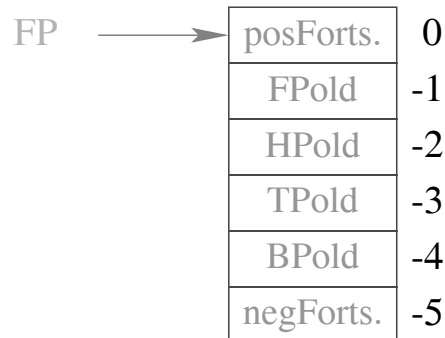
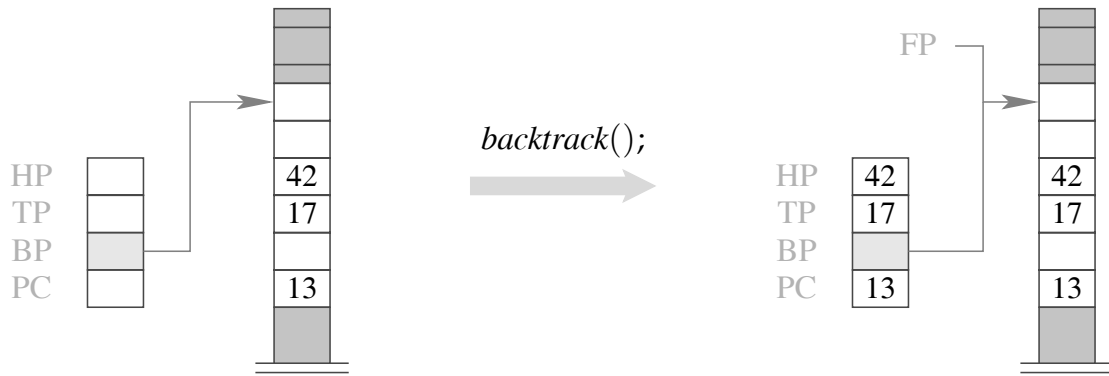


Abb. 4.27. Die organisatorischen Zellen der WiM.

erhalten die Relativadressen -3 und -2 .

Ein Aufruf der Laufzeitfunktion **void** *backtrack()* kehrt zum aktuellen Rücksetzpunkt zurück und fährt dort mit der negativen Fortsetzungsadresse fort (Abb. 4.28). Bevor jedoch die nächste Alternative angesprungen werden kann, müssen erst die seit der Wahl der letzten Alternative vorgenommenen Variablenbindungen rückgängig gemacht werden. Dies geschieht in zwei Schritten. Indem der Haldenzeiger auf seinen im Kellerrahmen des Rücksetzpunkts gesicherten Wert zurückgesetzt wird, werden sämtliche seitdem angelegten Haldenobjekte inklusive der darin enthaltenen Referenzen beseitigt. Wir wären damit bereits fertig, wären keine älteren Referenzen gebunden worden. Die Adressen der älteren Referenzobjekte *a*, die seit Anlegen des Rücksetzpunkts gebunden wurden, haben wir jedoch stets auf der Spur durch einen Aufruf *trail* (*a*) vermerkt. Sie sollten auf der Spur an den Adressen $S[FP - 3] + 1, S[FP - 3] + 2, \dots, TP$ stehen. Das *Entbinden* dieser Referenzobjekte erledigt der Aufruf *reset* ($S[FP - 3], TP$).

Die Funktionen **void** *trail* (**ref** *u*) und **void** *reset* (**ref** *y*, **ref** *x*) lauten deshalb:



```

void backtrack () {
    FP ← BP; HP ← S[FP - 2];
    reset (S[FP - 3], TP);
    TP ← S[FP - 3]; PC ← S[FP - 5];
}

```

Abb. 4.28. Die Laufzeitfunktion *backtrack()*.

```

void trail (ref u) {
    if (u < S[BP - 2]) {
        TP ← TP + 1;
        T[TP] ← u;
    }
}

void reset (ref x, ref y) {
    for (ref u ← y; x < u; u--)
        H[T[u]] ← (R, T[u]);
}

```

Die Kellerezelle $S[BP - 2]$ enthält den Haldenzeiger bei Anlegen des letzten Rücksetzpunkts. Der entsprechende Vergleich in der Funktion *trail()* erlaubt es uns deshalb auf einfache Weise zu entscheiden, ob wir eine Referenz auf der Spur vermerken sollen oder ob dies nicht notwendig ist.

4.7.2 Zusammenfügung

Nehmen wir an, das Prädikat q/k sei durch die Folge von Klauseln $rr \equiv r_1, \dots, r_f$ definiert, wobei nun $f > 1$ ist. Dann benötigen wir Code, um zuerst den Rücksetzpunkt zu *initialisieren*. Anschließend müssen wir dafür sorgen, dass die verschiedenen Alternativen der Reihe nach ausprobiert werden. Am Ende sollte dann der Rücksetzpunkt *aufgegeben* werden. Folglich erzeugen wir den folgenden Code:

```

codeP  $rr = q/k$  : setbtp
                    try  $A_1$ 
                    ...
                    try  $A_{f-1}$ 
                    delbtp
                    jump  $A_f$ 
 $A_1$  : codeC  $r_1$ 
      ...
 $A_f$  : codeC  $r_f$ 

```

Beachten Sie, dass der Rücksetzpunkt bereits *vor* Ausprobieren der letzten Alternative aufgegeben und diese letzte Möglichkeit direkt angesprungen wird. Dies ist sinnvoll, weil bei einem erneuten Fehlschlag zu dem gegenwärtigen Kellerrahmen nicht mehr zurückgekehrt werden muss.

Beispiel 4.7.1 Betrachten wir das folgende Prädikat:

$$\begin{aligned}
 s(X) &\Leftarrow t(\bar{X}) \\
 s(X) &\Leftarrow \bar{X} = a
 \end{aligned}$$

Die Übersetzung des Prädikats s liefert gemäß unserem Schema:

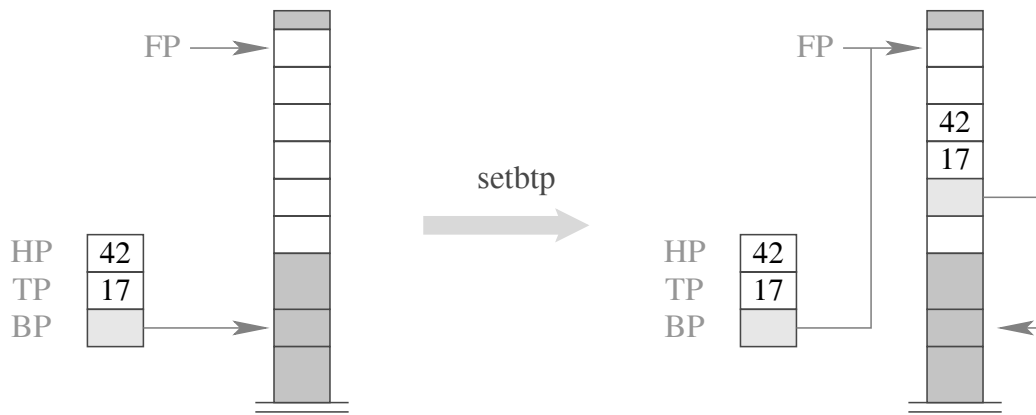
$s/1$:	setbtp	A :	pushenv 1	B :	pushenv 1
	try A		mark C		putref 1
	delbtp		putref 1		uatom a
	jump B		call $t/1$		popenv
		C :	popenv		

□

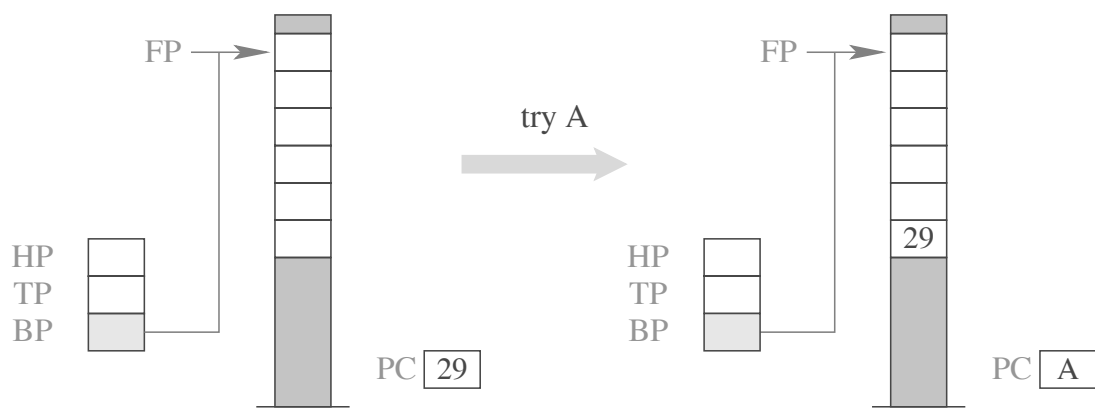
Für unser Übersetzungsschema benötigen wir einige neue Instruktionen. Die Instruktion **setbtp** legt einen Rücksetzpunkt an. Dazu rettet sie die Register HP , TP und BP (Abb. 4.29). Die Instruktion **try** A versucht die Alternative mit Anfangsadresse A und ersetzt die negative Fortsetzungsadresse im Kellerrahmen durch den aktuellen PC , d.h. der Adresse des nächsten Befehls in der *try*-Kette. (Abb. 4.30). Die Instruktion **delbtp** schließlich restauriert den alten Wert des Rücksetzzeigers (Abb. 4.31).

4.8 Die Endbehandlung von Klauseln

Jetzt fehlt für die Übersetzung von Klauseln und Prädikaten nur noch die Endbehandlung einer Klausel durch die Instruktion **popenv**. Erinnern wir uns an die Übersetzung von Klauseln:



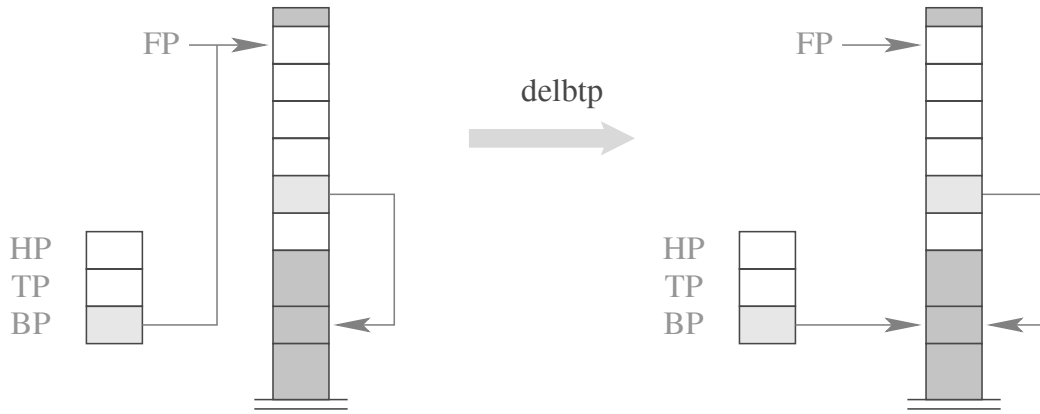
$$S[FP - 2] \leftarrow HP; S[FP - 3] \leftarrow TP;$$

$$S[FP - 4] \leftarrow BP; BP \leftarrow FP;$$
Abb. 4.29. Die Instruktion **setbtp**.

$$S[FP - 5] \leftarrow PC; PC \leftarrow A;$$
Abb. 4.30. Die Instruktion **try A**.

```
codeC r = pushenv m
           codeG g1 ρ
           ...
           codeG gn ρ
           popenv
```

Natürlich ist unser Ziel, am Ende einer Klausel den aktuellen Kellerrahmen aufzugeben. Dies ist jedoch nicht immer möglich. So könnte ja der aktuelle Rahmen



$$BP \leftarrow S[FP - 4];$$

Abb. 4.31. Die Instruktion **delbtp**.

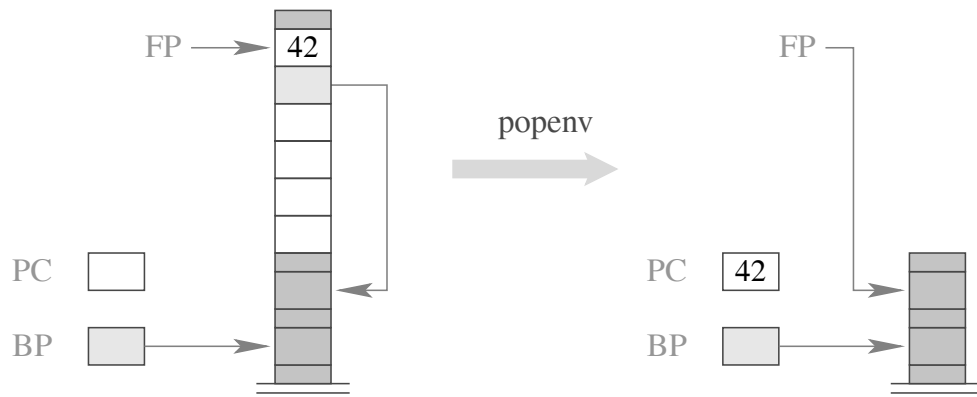
ein Rücksetzpunkt mit weiteren offenen Alternativen sein. Oder es wurden bei der Abarbeitung der Literale im Rumpf der Klausel Rücksetzpunkte angelegt, die nicht einfach vom Keller entfernt werden können. Dies sind tatsächlich auch die einzigen beiden Hindernisse für eine Aufgabe des aktuellen Kellerrahmens. Ein einfacher Test der Bedingung für die Rahmenfreigabe besteht darin, den aktuellen BP mit dem FP zu vergleichen. Gilt $BP < FP$, können wir getrost den Kellerrahmen aufgeben, andernfalls müssen wir ihn behalten.

Diesen Test führt die Instruktion **popenv** durch (Abb. 4.32 und 4.33). Zuerst versucht die Instruktion, den Kellerrahmen freizugeben. Dann restauriert sie die Register FP und PC . Beachten Sie, dass es der Instruktion **popenv** – obwohl dies ihr Name vielleicht suggerieren mag – nicht immer gelingt, den Kellerrahmen aufzugeben! Falls die Freigabe eines Kellerrahmens im Rumpf einer Klausel scheitert, werden damit also weitere Daten oben auf dem Keller allokiert. Bei Rückkehr zu dem Kellerrahmen der Klausel können die lokalen Variablen aber immer noch über den FP adressiert werden.

4.9 Anfragen und Programme

Damit haben wir Übersetzungsschemata für sämtliche Bestandteile eines ProL-Programms zur Verfügung. Bleibt uns also nur noch festzulegen, wie wir mit einem gesamten Programm p umgehen. Nehmen wir an, das Programm p bestehe aus den Prädikatsdefinitionen $rr_1 \dots rr_h$ sowie der Anfrage $\Leftarrow g_1, \dots, g_l$. Dann sollte der Code für das Programm p die folgenden Bestandteile enthalten:

- Code zur Auswertung der Anfrage $\Leftarrow g_1, \dots, g_l$,
- einer Instruktion **no** für Fehlschlag, sowie
- Code für die Prädikate rr_i .



$$\text{if } (BP < FP) \text{ } SP \leftarrow FP - 6;$$

$$FP \leftarrow S[FP - 1]; \text{ } PC \leftarrow S[FP];$$

Abb. 4.32. Die Instruktion **popenv** mit Freigabe des Kellerrahmens.

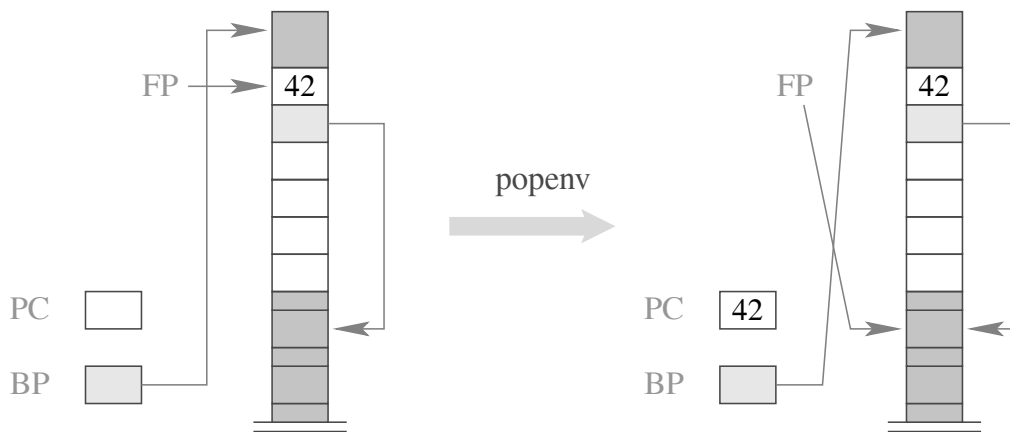


Abb. 4.33. Die Instruktion **popenv** ohne Freigabe des Kellerrahmens.

Wir nehmen dabei an, dass vor der Programmausführung $SP = FP = BP = TP = -1$ und $PC = HP = 0$ sind. Vor der Auswertung der Anfrage müssen die Register SP, FP und BP korrekt initialisiert und ein erster Kellerrahmen auf dem Keller angelegt werden. Danach muss die Ergebnissubstitution zurückgegeben (oder Fehlschlag gemeldet) werden:

```

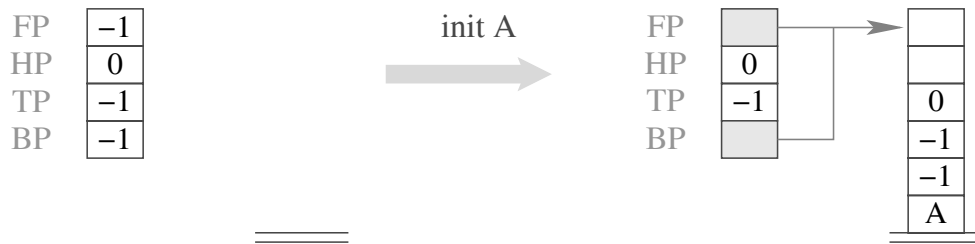
code  $p$  =   init  $A$ 
            pushenv  $d$ 
            codeG  $g_1$   $\rho$ 
            ...
            codeG  $g_l$   $\rho$ 
            halt  $d$ 
     $A$  : no
        codeP  $rr_1$ 
        ...
        codeP  $rr_h$ 

```

sofern $\{X_1, \dots, X_d\} = \text{free}(g_1, \dots, g_l)$ die Menge der Variablen in der Anfrage g_1, \dots, g_l ist und $\rho(X_i) = i, i = 1, \dots, d$ gilt.

Die Instruktion **halt** d beendet die Programmausführung und stellt die Bindungen der d Variablen der Welt zur Verfügung, bzw. löst Backtracking aus – sofern von der Benutzerin gefordert.

Die Instruktion **init** A initialisiert insbesondere die Register SP, FP und PC und vermerkt im Kellerrahmen die Adresse A als negative Fortsetzungsadresse (Abb. 4.34). An der Adresse A für das Fehlschlagen des Ziels haben wir die Instruktion **no**



```

 $SP \leftarrow FP \leftarrow BP \leftarrow 5;$ 
 $S[0] \leftarrow A; S[1] \leftarrow S[2] \leftarrow -1;$ 
 $S[3] \leftarrow 0;$ 

```

Abb. 4.34. Die Instruktion **init** A .

untergebracht, welche **no** auf die Standardausgabe schreibt und dann anhält.

Nun sind wir an einem Punkt angekommen, an dem wir ein erstes vollständiges ProL-Programm übersetzen können.

Beispiel 4.9.1 Wir nehmen dazu das folgende kleine Programm:

$$\begin{array}{lll}
 t(X) \Leftarrow \bar{X} = b & q(X) \Leftarrow s(\bar{X}) & s(X) \Leftarrow \bar{X} = a \\
 p \Leftarrow q(X), t(\bar{X}) & s(X) \Leftarrow t(\bar{X}) & \Leftarrow p
 \end{array}$$

Die Übersetzung liefert:

	init N		popenv	$q/1 :$	pushenv 1	$E :$	pushenv 1
	pushenv 0	$p/0 :$	pushenv 1		mark D		mark G
	mark A		mark B		putref 1		putref 1
	call $p/0$		putvar 1		call $s/1$		call $t/1$
$A :$	halt 0		call $q/1$	$D :$	popenv	$G :$	popenv
$N :$	no	$B :$	mark C	$s/1 :$	setbtp	$F :$	pushenv 1
$t/1 :$	pushenv 1		putref 1		try E		putref 1
	putref 1		call $t/1$		delbtp		uatom a
	uatom b	$C :$	popenv		jump F		popenv

□

4.10 Optimierung I: Letzte Ziele

Ähnlich wie bei funktionalen Programmen lässt sich die Effizienz der Implementierung durch einfache Optimierungen beträchtlich steigern. Betrachten wir etwa das Prädikat **app** aus der Einleitung:

$$\begin{aligned} \text{app}(X, Y, Z) &\Leftarrow X = [], Y = Z \\ \text{app}(X, Y, Z) &\Leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \end{aligned}$$

Offenbar ist der rekursive Aufruf das *letzte* Ziel in der Klausel. Nach (erfolgreicher) Beendigung dieses Ziels kehrt man nicht mehr zum gegenwärtigen Aufrufer zurück. Ein solches Ziel wollen wir darum wie einen letzten Aufruf bei der Übersetzung funktionaler Programme behandeln, nach Möglichkeit wird kein neuer Kellerrahmen dafür allokiert, sondern es wird im *aktuellen* Kellerrahmen ausgewertet.

Betrachten wir eine Klausel r der Form:

$$q_0(X_1, \dots, X_k) \Leftarrow g_1, \dots, g_n$$

mit m lokalen Variablen, wobei das letzte Ziel g_n das Literal $q(t_1, \dots, t_h)$ ist. Das Zusammenspiel zwischen **code_C** und **code_G** wollen wir uns etwas genauer ansehen:

code_C $r =$	pushenv m
	code_G $g_1 \rho$
	...
	code_G $g_{n-1} \rho$
	mark B
	code_A $t_1 \rho$
	...
	code_A $t_h \rho$
	call q/h
$B :$	popenv

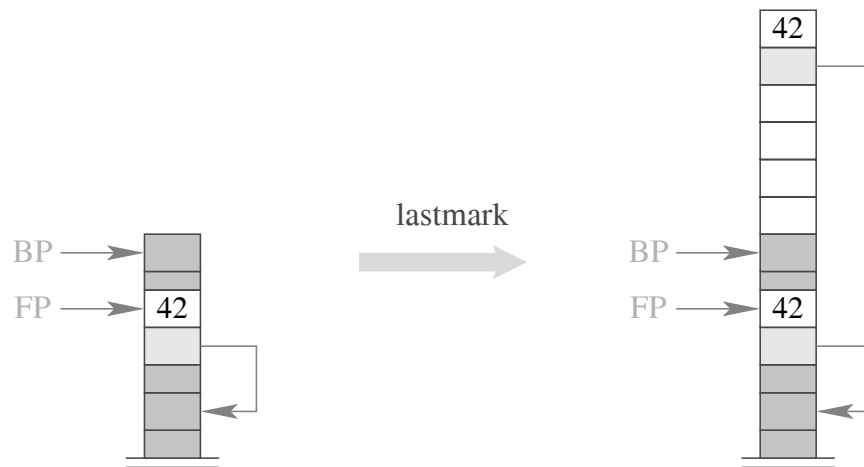
Als Optimierung ersetzen wir die Instruktion **mark** B durch eine neue Instruktion **lastmark** und die Folge **call** q/h ; **popenv** durch die neue Instruktion **lastcall** q/h m . Dadurch erhalten wir:

```

codeC  $r =$  pushenv  $m$ 
               codeG  $g_1$   $\rho$ 
               ...
               codeG  $g_{n-1}$   $\rho$ 
               lastmark
               codeA  $t_1$   $\rho$ 
               ...
               codeA  $t_h$   $\rho$ 
               lastcall  $q/h$   $m$ 

```

Falls die gegenwärtige Klausel nicht die letzte ist oder die g_1, \dots, g_{n-1} Rücksetzpunkte erzeugt haben, ist $FP \leq BP$. Dann können wir den aktuellen Rahmen vor dem letzten Aufruf noch nicht aufgeben. Stattdessen muss die Instruktion **lastmark** einen neuen Rahmen mit einer Referenz auf den Vorgänger anlegen (Abb. 4.35). Falls dagegen $FP > BP$, dann kann der aktuelle Kellerrahmen für den letzten



$$\text{if } (FP \leq BP) \{ SP \leftarrow SP + 6; S[SP] \leftarrow S[FP]; S[SP - 1] \leftarrow FP; \}$$

Abb. 4.35. Die Instruktion **lastmark**.

Aufruf verwendet werden. Die Instruktion **lastmark** tut dann nichts.

Entsprechend vergleicht auch die Instruktion **lastcall** q/h m die Register FP und BP (Abb. 4.36). Falls $FP \leq BP$, verhält sich **lastcall** q/h m wie ein normales **call** q/h . Falls dagegen $FP > BP$, werden die aktuellen Parameter in den Zellen $S[FP + 1], S[FP + 2], \dots, S[FP + h]$ ausgetauscht und q/h angesprungen.

Die Differenz zwischen den alten und neuen Adressen der verschobenen aktuellen Parameter ist $SP - h - FP = m$, d.h. ergibt gerade die Anzahl der lokalen Variablen im Kellerrahmen.

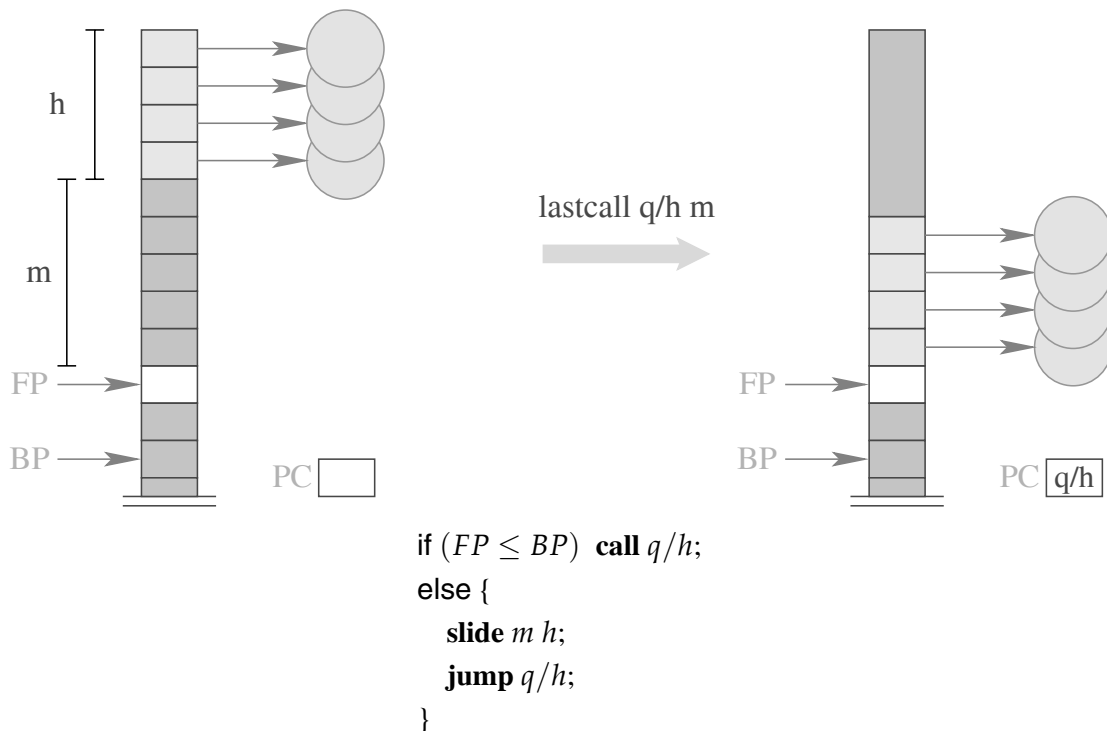


Abb. 4.36. Die Instruktion **lastcall** $q/h\ m$.

Beispiel 4.10.1 Betrachten wir die Klausel r :

$$a(X, Y) \Leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Die Optimierung letzter Ziele liefert für `codeC r` in der Adressumgebung $\{X \mapsto 1, Y \mapsto 2, X_1 \mapsto 3\}$:

	mark A	A :	lastmark
pushenv 3	putref 1		putref 3
	putvar 3		putref 2
	call $f/2$		lastcall $a/2\ 3$

□

Ist das letzte Literal der letzten Klausel gleichzeitig das *einzige* Literal in dieser Klausel, können wir auf **lastmark** ganz verzichten und auch **lastcall** $q/h\ m$ zu der unbedingten die Folge **slide** $m\ h$; **jump** q/h vereinfachen.

Beispiel 4.10.2 Betrachten Sie die letzte Klausel des Prädikats **app**:

$$\text{app}(X, Y, Z) \Leftarrow \bar{X} = [H|X'], \bar{Z} = [\bar{H}|Z'], \text{app}(\bar{X}', \bar{Y}, \bar{Z}')$$

Hier ist das letzte Ziel das einzige Literal. Folglich erhalten wir:

A :	pushenv 6		uref 4	bind
	putref 1	B :	putvar 4	son 2
	ustruct [[]/2 B		putvar 5	E :
	son 1		putstruct [[]/2	putref 5
	uvar 4		up E	putref 6
	son 2	C :	check 4	slide 6 3
	uvar 5		putref 4	jump app/3
	up C		putvar 6	
			putstruct [[]/2	

Für das Prädikat **app** haben wir es also tatsächlich geschafft, die Rekursion durch den letzten Aufruf in eine Schleife zu übersetzen. \square

4.11 Optimierung II: Verkleinerung von Kellerrahmen

Weil Kellerrahmen viel häufiger auf dem Keller gehalten werden müssen als etwa bei der MaMa, leiden Implementierungen logischer Sprachen unter erhöhtem Kellerplatzverbrauch. Jede Möglichkeit, systematisch Platz auf dem Keller einzusparen, ist deshalb hoch willkommen. Eine Idee in diese Richtung besteht darin, die lokalen Variablen einer Klausel gemäß ihrer *Lebensdauer* zu ordnen. Wird eine Variable im weiteren Verlauf der Abarbeitung der Klausel nicht mehr explizit verwendet, nennen wir sie *tot*. Tote Variablen können wir, wenn immer möglich, wieder vom Keller entfernen.

Beispiel 4.11.1 Betrachten Sie die folgende Klausel:

$$a(X, Z) \Leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

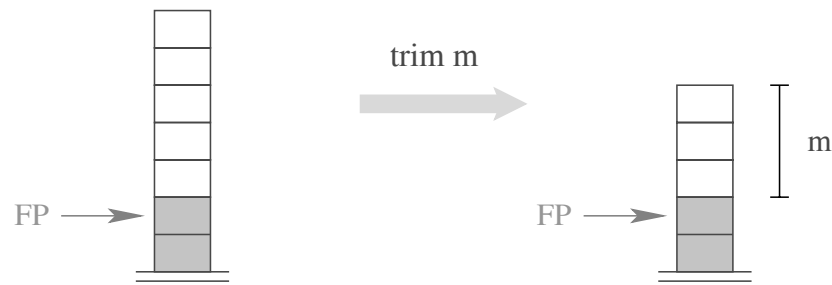
Hinter dem Literal $p_2(\bar{X}_1, X_2)$ ist die Variable X_1 tot. Hinter dem Literal $p_3(\bar{X}_2, X_3)$ ist die Variable X_2 ebenfalls tot. \square

Nehmen wir an, wir hätten ein nichtletztes Ziel, nach dem die ersten m Variablen lebendig sind. Dann können wir eine Instruktion **trim** m einfügen (Abb. 4.37). Die toten lokalen Variablen dürfen aber nur eliminiert werden, wenn keine Rücksetzpunkte angelegt wurden, deren erneute Auswertung einen späteren Zugriff auf die toten Variablen erfordert. Deshalb vergleicht die Instruktion **trim** m die Register FP und BP und bereinigt die toten Variablen nur, wenn $FP \geq BP$ gilt.

Beispiel 4.11.2 Betrachten Sie erneut die Klausel aus Beispiel 4.11.1:

$$a(X, Z) \Leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

Gemäß der Lebensdauer ergibt sich folgende Anordnung der Variablen:



$\text{if } (FP \geq BP) \ SP \leftarrow FP + m;$

Abb. 4.37. Die Instruktion **trim** m .

$$\rho = \{X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_2 \mapsto 4, X_1 \mapsto 5\}$$

Die entsprechende Einfügung der *trim*-Instruktionen liefert uns:

pushenv 5 mark A putref 1 putvar 5 call p₁/2	A : B :	mark B putref 5 putvar 4 call p₂/2 trim 4	mark C putref 4 putvar 3 call p₃/2 trim 3	lastmark putref 3 putref 2 lastcall p₄/2 3
--	--	---	---	--

□

4.12 Optimierung III: Klausel-Indizierung

Oft werden Prädikate durch Fallunterscheidung nach einem, in der Regel dem ersten Argument definiert. Die Berücksichtigung dieses Arguments kann viele Alternativen sehr schnell ausschließen! Die Vorteile einer entsprechenden Implementierung liegen auf der Hand: das Fehlschlagen eines Prädikats wird früher entdeckt. Die Rücksetzpunkte werden früher beseitigt. Die Kellerrahmen werden somit früher aufgegeben.

Beispiel 4.12.1 Die Situation können wir uns erneut an unserem **app**-Prädikat verdeutlichen:

$$\begin{aligned} \text{app}(X, Y, Z) &\Leftarrow X = [], Y = Z \\ \text{app}(X, Y, Z) &\Leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \end{aligned}$$

Wir beobachten:

- Falls der Wurzelkonstruktor $[]$ ist, ist nur die erste Klausel anwendbar.
- Falls der Wurzelkonstruktor $[|]$ ist, ist nur die zweite Klausel anwendbar.
- Jeder andere Wurzelkonstruktor sollte *fehlschlagen*.

- Nur wenn das erste Argument eine ungebundene Variable ist, müssen beide Alternativen probiert werden.

□

Die Idee besteht deshalb darin, getrennte *try*-Ketten für jeden möglichen Konstruktor einzuführen. In dieser Darstellung berücksichtigen wir der Einfachheit halber den Wurzelknoten des ersten Arguments. Abhängig vom Ergebnis führen wir einen *indizierten* Sprung in die entsprechende Kette durch.

Nehmen wir an, das Prädikat q/k sei durch die Folge rr von Klauseln $r_1 \dots r_m$ definiert. Sei $tchains\ rr$ die Folge der *try*-Ketten entsprechend den Wurzelkonstruktoren in Unifikationen $X_1 = t$ am Anfang des Rumpfs der Klauseln. Hier verzichten wir auf eine formale Spezifikation, wie die *try*-Ketten aus der Folge rr der das Prädikat q/k definierenden Klauseln gewonnen werden können. Stattdessen illustrieren wir die Idee anhand eines Beispiels.

Beispiel 4.12.2 Betrachten wir erneut das **app**-Prädikat und nehmen an, der Code der beiden Klauseln beginne an den Adressen A_1 und A_2 . Dann erhalten wir die folgenden vier Ketten:

```
VAR:   setbtp    // Variablen   NIL:       jump A1    // Atom [ ]
       try A1
       delbtp
       jump A2
CONS:   jump A2    // Konstruktor [[]]
DEFAULT: fail      // Default
```

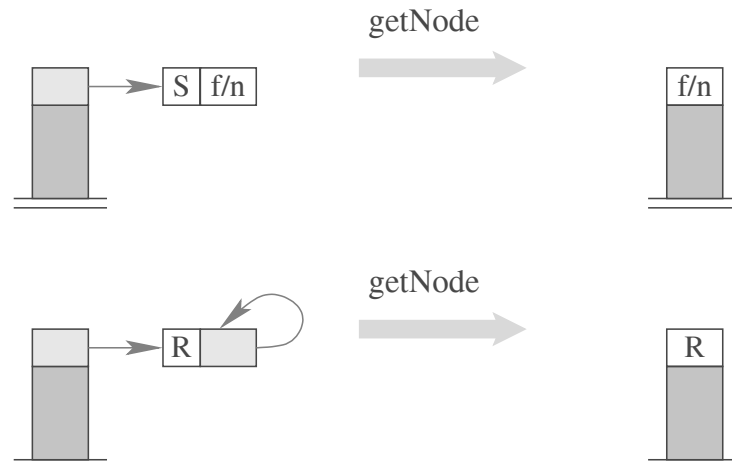
Die neue Instruktion **fail** ist für alle Konstruktoren außer $[]$ und $[[]]$ zuständig:

fail = *backtrack()*

Ihre Ausführung löst direkt Backtracking aus. □

Mithilfe der Folge $tchains\ rr$ zu dem Prädikat q/k lässt sich nun leicht der Code für das Prädikat q/k erzeugen:

```
codeP rr    =    putref 1
                 getNode    // extrahiert die Wurzel-Beschriftung
                 index q/k    // springt zum try-Block
                 tchains rr
A1 :    codeC r1
...
Am :    codeC rm
```



```

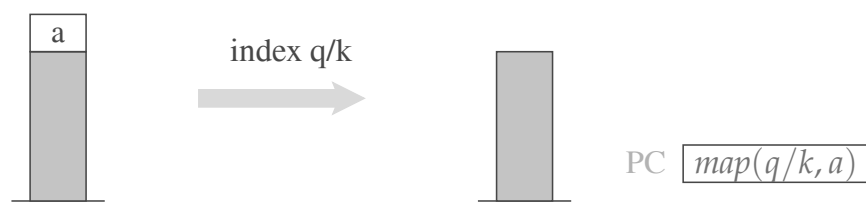
switch (H[S[SP]]) {
  case (A, a) : S[SP] ← a; break;
  case (S, f/n) : S[SP] ← f/n; break;
  case (R, _) : S[SP] ← R;
}

```

Abb. 4.38. Die Instruktion **getNode**.

Die Instruktion **getNode** liefert R , falls der Verweis oben auf dem Keller auf eine ungebundene Variable zeigt. Andernfalls liefert sie den Inhalt des Haldenobjekts (Abb. 4.38).

Die Instruktion **index** q/k benutzen wir, um einen indizierten Sprung an die entsprechende *try*-Kette durchzuführen (Abb. 4.39).



$$PC \leftarrow \text{map}(q/k, S[SP]); SP--;$$

Abb. 4.39. Die Instruktion **index** q/k .

Die Funktion *map()* in der Definition der Instruktion **index** q/k liefert zu gegebenem Prädikat und Knoteninhalt die Startadresse der entsprechenden *try*-Kette. Wir

lassen offen, wie diese Funktion implementiert werden könnte. Sicherlich bietet sich aber eine Implementierung mithilfe von Hash-Tabellen an.

4.13 Erweiterung: Der Cut Operator

Wirkliches Prolog stellt zusätzlich einen Operator „!“ zur Verfügung, den *Cut*. Dieser Operator erlaubt es, den Suchraum durch Beschränken des Rücksetzens explizit zu beschneiden. Betrachten Sie etwa das folgende Prädikat:

$$\begin{aligned}\text{branch}(X, Y) &\Leftarrow p(\bar{X}), !, q_1(\bar{X}, \bar{Y}) \\ \text{branch}(X, Y) &\Leftarrow q_2(\bar{X}, \bar{Y})\end{aligned}$$

Sobald die Anfragen *vor* dem Cut erfolgreich waren, sind alle dabei getroffenen Auswahlen festgelegt (committed). Backtracking wird nur noch zu Rücksetzpunkten *vor* der Abarbeitung der linken Seite zurückkehren. Dieses operationale Verhalten wollen wir nun implementieren. Bei einem Vorkommen des Cut-Operators sollte darum der Vorgänger des *BP* des aktuellen Kellerrahmens restauriert werden. Anschließend werden alle Kellerrahmen oberhalb der lokalen Variablen beseitigt, da man zu keinem dieser Rücksetzpunkte zurückkehren wird. Folglich übersetzen wir den Cut in die Folge:

prune
pushenv *m*

wobei *m* die Anzahl der (noch benötigten) lokalen Variablen der Klausel ist.

Beispiel 4.13.1 Betrachten wir unser Beispiel:

$$\begin{aligned}\text{branch}(X, Y) &\Leftarrow p(\bar{X}), !, q_1(\bar{X}, \bar{Y}) \\ \text{branch}(X, Y) &\Leftarrow q_2(\bar{X}, \bar{Y})\end{aligned}$$

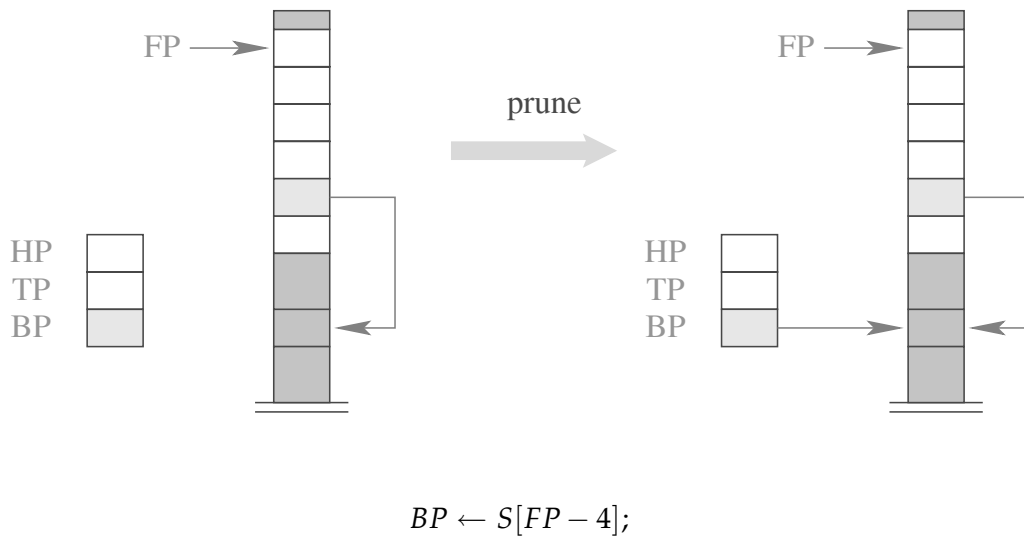
Eine optimierte Übersetzung liefert hier:

setbtp	A :	pushenv 2	C :	prune	putref 1	B :	pushenv 2
try A		mark C		pushenv 2	putref 2		putref 1
delbtp		putref 1			slide 2 2		putref 2
jump B		call p/1			jump q ₁ /2		slide 2 2
							jump q ₂ /2

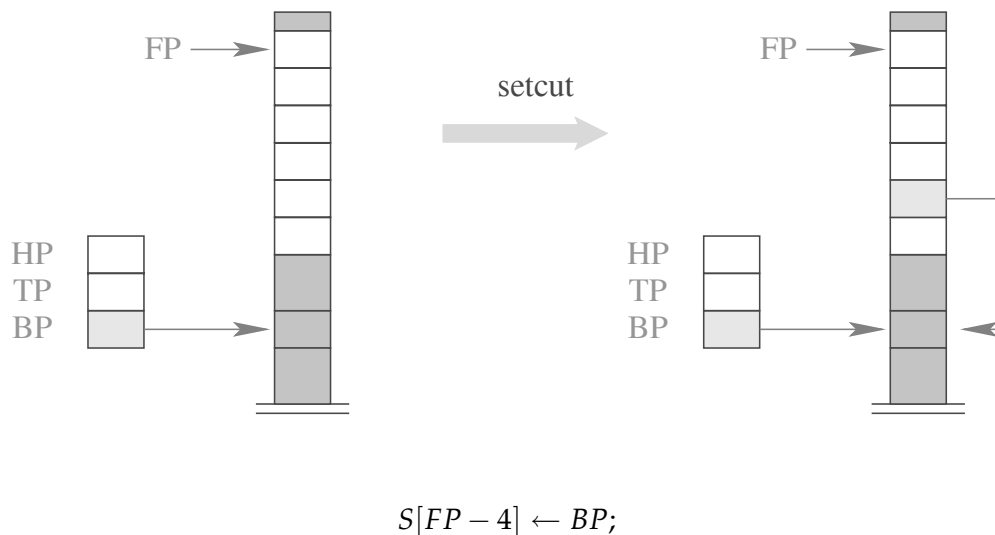
□

Die neue Instruktion **prune** restauriert den *BP* (Abb. 4.40).

Damit bleibt aber noch ein Problem: Ist eine Klausel *einzel*n in einer *try*-Kette, dann haben wir (zumindest bisher) den *BP* vor Betreten des Aufrufs nicht innerhalb des Kellerrahmens abgelegt. Folglich kann damit die Instruktion **prune** den *BP* auch nicht sinnvoll restaurieren.

Abb. 4.40. Die Instruktion **prune**.

Damit der Cut auch für Prädikate mit einer einzigen Klausel bzw. für *try*-Ketten der Länge 1 gilt, fügen wir eine zusätzliche Instruktion **setcut** vor dem Code (bzw. dem unbedingten Sprung an den Anfang des Codes) jeder solchen einzelnen Klausel ein, die einen Cut enthält. Die neue Instruktion **setcut** rettet gezielt den aktuellen Wert des *BP*.

Abb. 4.41. Die Instruktion **setcut**.

Damit kommen wir zum letzten Beispiel dieses Kapitels: *Negierung durch Fehlschlag* (Negation by Failure). Die Programmiersprache Prolog orientiert sich zwar

an prädikatenlogischen Formeln, soweit sie sich durch Horn-Klauseln beschreiben lassen. Exakte logische Negation lässt sich in Prolog jedoch nicht ausdrücken. Stattdessen bietet Prolog Negierung durch Fehlschlag an. Diese lässt sich mithilfe des Cut-Operators implementieren.

Beispiel 4.13.2 Das folgende Prädikat notQ/k soll immer dann erfolgreich sein, wenn q/k fehlschlägt:

$$\begin{aligned}\text{notQ}(X_1, \dots, X_k) &\Leftarrow q(\bar{X}_1, \dots, \bar{X}_k), !, \text{fail} \\ \text{notQ}(X_1, \dots, X_k) &\Leftarrow\end{aligned}$$

wobei das Ziel fail immer fehlschlägt. Dann erhalten wir für notQ/k die folgende Folge von WiM-Instruktionen:

setbtp	<i>A</i> :	pushenv <i>k</i>	<i>C</i> :	prune	<i>B</i> :	pushenv <i>k</i>
try <i>A</i>		mark <i>C</i>		fail		popenv
delbtp		putref 1		popenv		
jump <i>B</i>		...				
		putref <i>k</i>				
		call q/k				

Dabei hätten wir in der Spalte ab Marke *C* die Instruktion **popenv** weglassen können. Auch die Instruktion **pushenv** *k* in der letzten Spalte ist überflüssig. \square

4.14 Exkurs: Speicherbereinigung

Sowohl bei der Ausführung eines MaMa- wie eines WiM-Programms können Objekte in der Halde auftreten, auf die es keine Verweise mehr gibt. Diese Objekte heißen *Müll* (Garbage), weil sie die weitere Programmausführung nicht mehr beeinflussen können. Ihr Speicherplatz sollte deshalb freigegeben und für das Anlegen anderer Objekte wiederverwendet werden.

Beachten Sie, dass unsere virtuelle Maschine WiM für logische Programmiersprachen tatsächlich bereits über eine Art Speicherplatz-Freigabe verfügt. Diese Speicherplatz-Freigabe betrifft jedoch nur den Platz, den *fehlgeschlagene Alternativen* belegt haben. Sie sollten sich jedoch davon überzeugen, dass auch bei bisher erfolgreicher Abarbeitung von Zielen nicht erreichbare Haldenobjekte anfallen können.

Keine praktische Implementierung einer modernen funktionalen oder logischen Programmiersprache kommt ohne ein Verfahren zur automatischen *Speicherbereinigung* (Garbage Collection) aus. Automatische Speicherbereinigung hat auch Eingang gefunden in die objektorientierte Programmiersprache Java.

In den letzten Jahren wurden eine Reihe interessanter, effizienter Verfahren für diese Aufgabe entwickelt, praktisch erprobt und verfeinert. Einen generellen Überblick geben etwa [Wil92, JL96]. Speicherbereinigung für Prolog betrachtet u.a.

[VSD02]. Hier stellen wir nur das allereinfachste Verfahren vor: eine *kopierende* Speicherbereinigung. Insbesondere wollen wir diskutieren, wie sich die Speicherbereinigung für die virtuelle Maschine MaMa von derjenigen für die virtuelle Maschine WiM unterscheidet.

Konzeptionell teilt eine kopierende Speicherbereinigung den Speicherbereich für die Halde in zwei Teile, von denen jeweils nur die eine Hälfte benutzt wird. Ist die benutzte Hälfte voll, werden alle *lebendigen* Objekte in die unbenutzte Hälfte übertragen, um anschließend die Rollen der beiden Hälften zu vertauschen.

Lebendig sind dabei alle Objekte, die zur weiteren Programmausführung noch benötigt werden. Dazu rechnen wir sämtliche Objekte, auf die Referenzen im Keller liegen. Weiterhin wird jedes Objekt als lebendig aufgefasst, auf das ein anderes lebendiges Objekt eine Referenz enthält.

Die Menge aller lebendigen Objekte lässt sich deshalb mit einem entsprechend angepassten Grapherreichbarkeits-Algorithmus ermitteln (Abb. 4.42). Im Beispiel

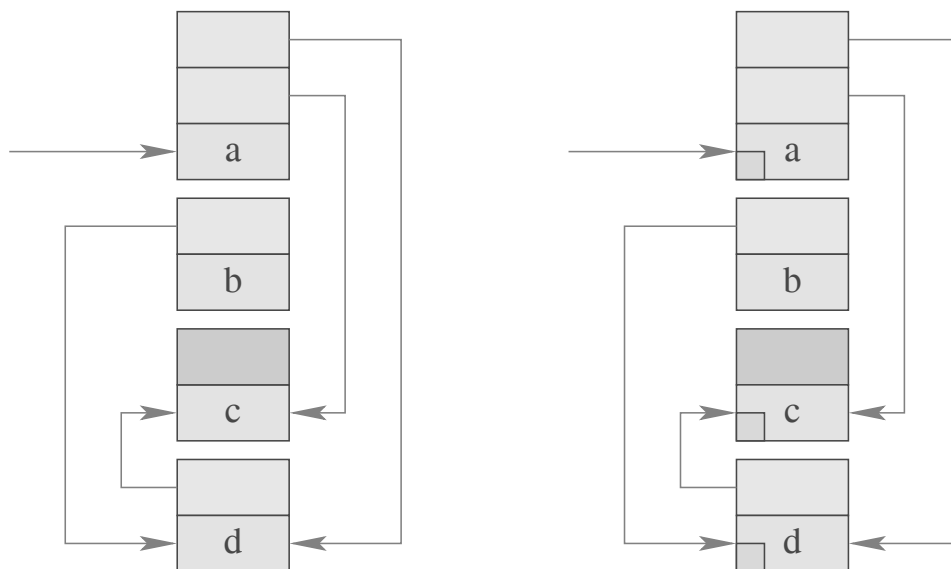


Abb. 4.42. Bestimmung der lebendigen Haldenobjekte.

haben wir dazu jedes Objekt mit einem Bit ausgestattet, das nach der Lebendigkeitsanalyse anzeigt, ob das Objekt bereits besucht wurde oder nicht.

Im nächsten Schritt werden die lebendigen Objekte in die frische Hälfte des Speichers kopiert. Diese Phase könnte tatsächlich mit der Erreichbarkeitsanalyse in der ersten Phase verschränkt werden. Beachten Sie jedoch, dass bei einfachem Kopieren der Objekte die Referenzen der Objekte nach wie vor in die alte Hälfte des Speichers zeigen. Deshalb platzieren wir an der alten Stelle jedes kopierten Objekts eine *Vorwärtsreferenz* auf die Stelle des kopierten Objekts in der neuen Hälfte (Abb. 4.43).

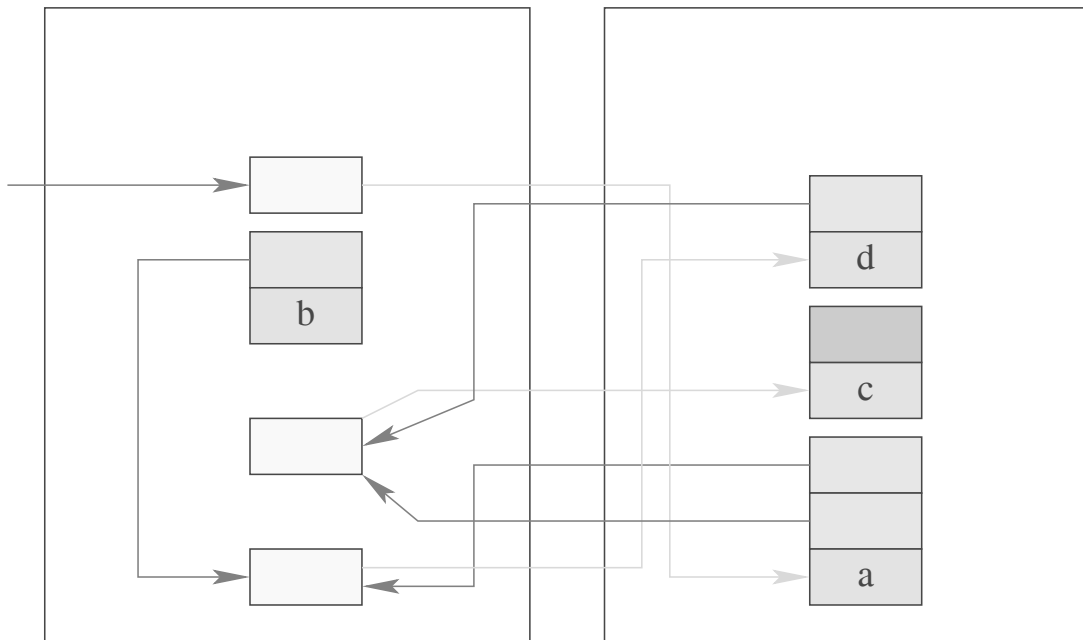


Abb. 4.43. Der Speicher nach Kopieren der lebendigen Haldenobjekte.

Nach dieser Phase zeigen alle Verweise der kopierten Objekte auf die Vorwärts-Referenzen in der alten Hälfte des Speichers. Nun müssen (zumindest konzeptuell) sowohl der Keller wie alle kopierten Objekte ein weiteres Mal besucht werden, um die Verweise auf die kopierten Objekte zu korrigieren (Abb. 4.44). Beachten Sie hier,

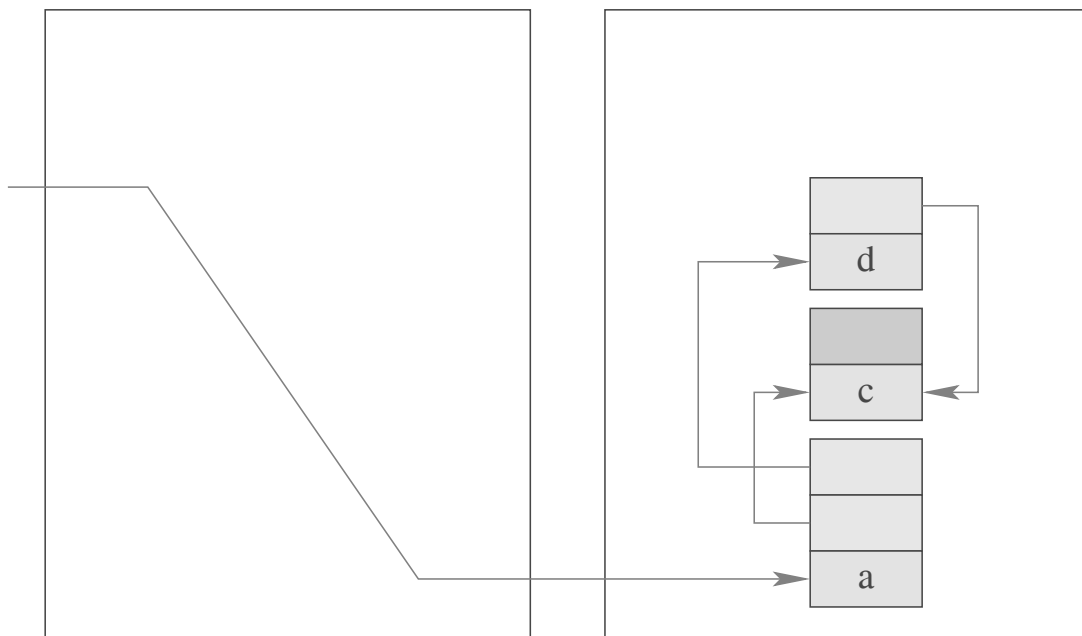


Abb. 4.44. Korrektur der Verweise auf kopierte Objekte.

dass die Verweise auf dem Keller auch direkt beim Kopieren der lebendigen Haldenobjekte korrigiert werden können. Das Gleiche gilt für Verweise in Haldenobjekten auf solche anderen Haldenobjekte, die bereits kopiert wurden.

In einem letzten Schritt müssen nun nur noch die Rollen von alter und neuer Speicherhälfte ausgetauscht werden.

Das bisher beschriebene Verfahren kann prinzipiell zur Speicherbereinigung bei der virtuellen Maschine MaMa für funktionale Programmiersprachen verwendet werden. Für die Speicherbereinigung der virtuellen Maschine WiM dagegen ist dieses Verfahren ungeeignet.

Der Grund ist, dass die Speicherbereinigung für die WiM mit dem Zurücksetzen harmonisieren muss. Das heißt, dass sich die *relative Lage der Haldenobjekte* während des Kopierens nicht verändern darf. In unserem Beispiel jedoch drehte sich die Reihenfolge der Haldenobjekte gerade um! Auch müssen im Anschluss an die Speicherbereinigung auch die Verweise in der Spur auf die neuen Objekte umgesetzt werden bzw. aus der Spur entfernt werden, wenn die betreffenden Objekte nicht mehr erreichbar sind.

Das bedeutet für unser Speicherbereinigungsverfahren, dass wir die als lebendig identifizierten Objekte nicht sofort kopieren dürfen (Abb. 4.45). Nachdem wir die lebendigen Haldenobjekte markiert haben, traversieren wir in einer zweiten Phase die alte Speicherhälfte, um nacheinander in genau der Reihenfolge, wie wir auf lebendige Objekte treffen, diese in die neue Hälfte des Speichers zu kopieren (Abb. 4.46). Anschließend müssen wir selbstverständlich erneut die Verweise auf die kopierten

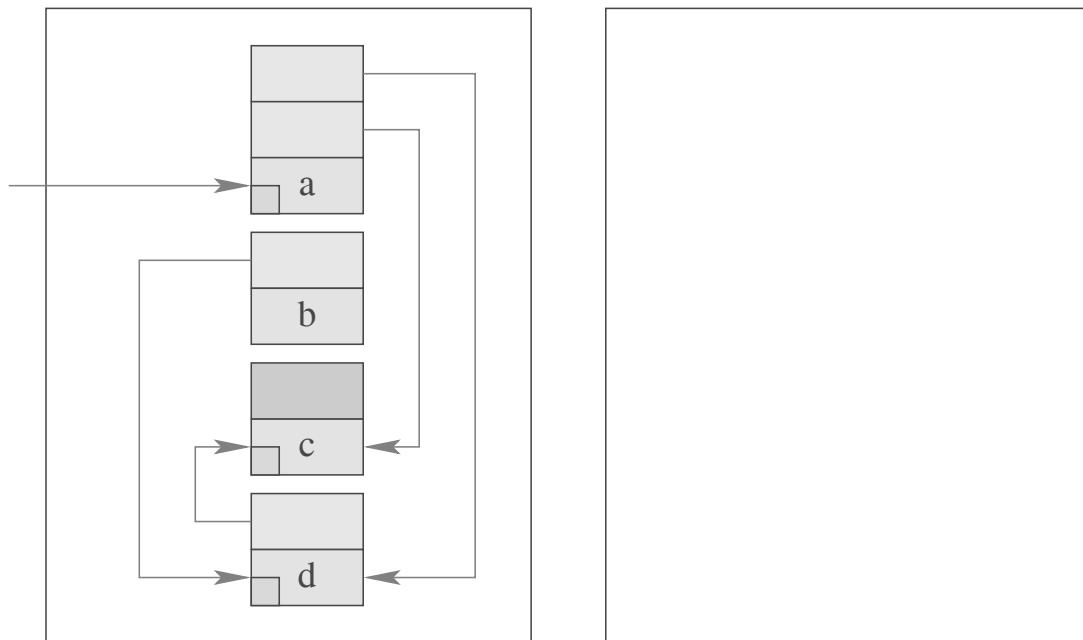


Abb. 4.45. Das Markieren der lebendigen Haldenobjekte.

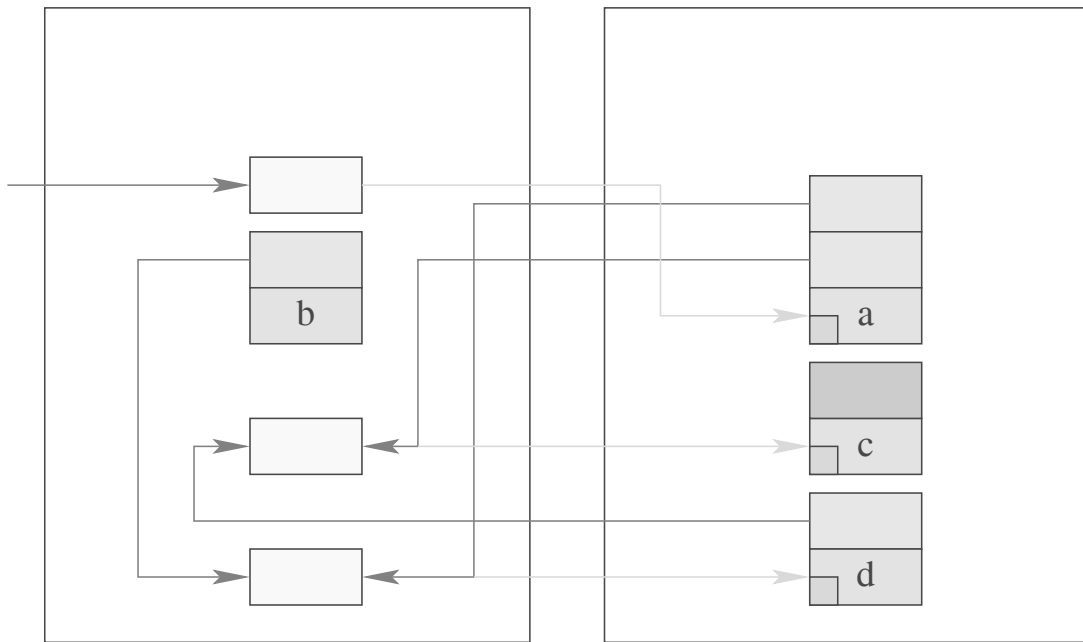


Abb. 4.46. Das modifizierte Kopieren der lebendigen Haldenobjekte.

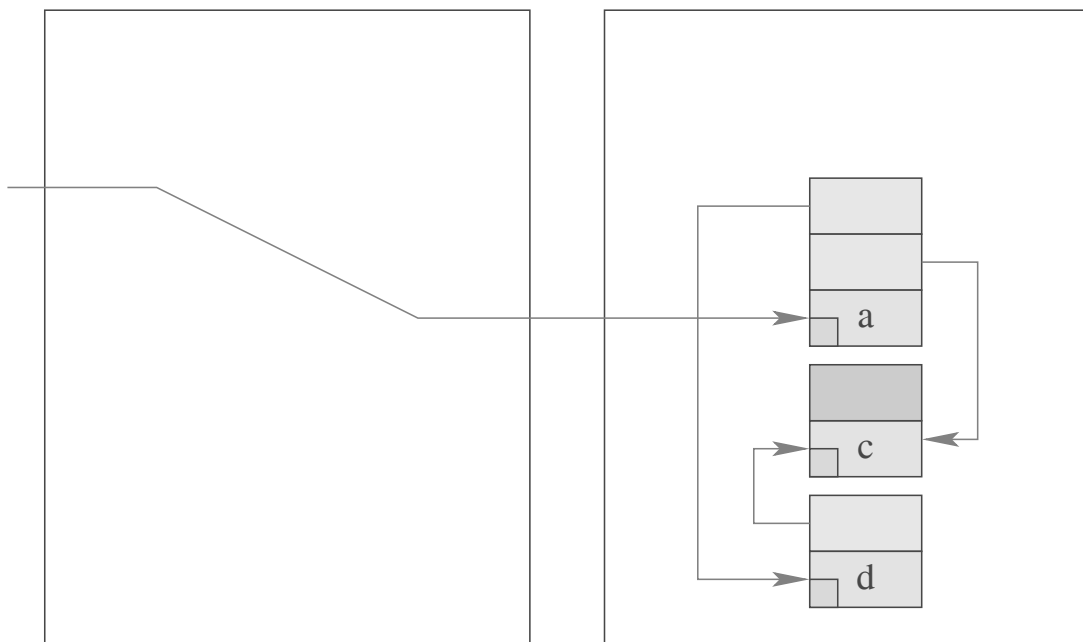


Abb. 4.47. Die abschließende Korrektur der Referenzen.

Haldenobjekte korrigieren (Abb. 4.47).

Aus dieser Darstellung konnten Sie ersehen, dass wir bei der Speicherbereinigung für die WiM aufpassen mussten, die Randbedingungen für die Korrektheit unserer Codeerzeugung einzuhalten. Das resultierende Verfahren ist darum auch nicht so effizient wie dasjenige für die funktionale Maschine: das liegt unter anderem daran, dass es in der Kopier-Phase nicht mehr ausreicht, alleine die lebendigen Haldenobjekte zu besuchen. Vielmehr muss die gesamte alte Speicherhälfte traversiert werden (inklusive der nicht erreichbaren Objekte), um die richtige Anordnung der Objekte nach dem Kopieren zu gewährleisten.

4.15 Aufgaben

1. *Listen in Prolog.* Implementieren Sie folgende Prädikate in Prolog:
 - **last/2**, so dass der erste Parameter eine Liste und der zweite das letzte Element dieser Liste ist. Z.B. gilt:

$$\text{last}([1, 2, 3], 3)$$

- **reverse/2** mit zwei Listen in umgekehrter Elementreihenfolge. Z.B. sollte gelten:

$$\text{reverse}([1, 2, 3, 4], [4, 3, 2, 1])$$

- **chain/2** mit zwei Listen, von denen die zweite als zusammenhängende Teilliste in der ersten enthalten ist. Z.B.:

$$\text{chain}([1, 2, 3, 4, 5, 6], [2, 3, 4])$$

- **remove/3** mit einem Wert und zwei Listen als Parameter. Dabei soll die zweite Liste bis auf Entfernung aller Vorkommen des ersten Parameters identisch zur ersten Liste sein. Z.B.:

$$\text{remove}(2, [1, 2, 3, 2, 5], [1, 3, 5])$$

Hinweis: Sie dürfen selbstverständlich auch Hilfsprädikate einführen.

2. *Programmieren in ProL.* Betrachten Sie folgendes ProL-Programm:

$$\begin{aligned} \text{edge}(X, Y) &\Leftarrow X = a, Y = b \\ \text{edge}(X, Y) &\Leftarrow X = b, Y = a \\ \text{edge}(X, Y) &\Leftarrow X = c, Y = c \\ \text{reachable}(X, Y) &\Leftarrow X = Y \\ \text{reachable}(X, Y) &\Leftarrow \text{edge}(X, Z), \text{reachable}(Z, Y) \\ &\Leftarrow \text{reachable}(a, c) \end{aligned}$$

Wie verhält sich dieses Programm bei der Ausführung? Erklären Sie dieses Verhalten!

3. *Übersetzung von Termen und Zielen.* Generieren Sie code_A und code_G für folgende Terme/Ziele!

- $f(X, g(b, Y), g(\bar{X}, \bar{Z}))$
- $f(g(X, h(\bar{Y}, _), b), Z)$

Verwenden Sie dabei folgende Adressumgebung:

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$$

4. *Unifikation.* Überprüfen Sie, ob die folgenden Terme unifzierbar sind. Falls ja, geben Sie allgemeinste unifzierende Substitutionen an.

- $z(a(b(D)), d(e(F)), g(H))$ und $z(H, K, g(F))$
- $p(f(g(X)), Y, X)$ und $p(Z, h(X), i(Z))$
- $f(A, g(A, B))$ und $f(g(B, C), g(g(h(t), B), h(t)))$
- $a(b, X, d(e, Y, g(h, i, Z)))$ und $a(U, c, d(V, f, g(W, i, j)))$

5. *Unifikation: Laufzeit.* Beweisen Sie, dass die Laufzeit für die Unifikation von zwei Termen exponentiell zur Anzahl der vorkommenden Variablen sein kann!

Hinweis: Betrachten Sie die folgenden beiden Terme:

$$\begin{aligned} t_1 &= f(X_0, X_1, \dots, X_n) \\ t_2 &= f(b, a(X_0, X_0), a(X_1, X_1), \dots, a(X_{n-1}, X_{n-1})) \end{aligned}$$

6. *Die Deref-Funktion.* Die Laufzeitfunktion *deref* verkürzt Referenzketten.
- In welchen Fällen können Referenzketten entstehen, so dass *deref* mindestens einmal rekursiv aufgerufen werden muss? Geben Sie ein Beispiel an!
 - Wie lange können Referenzketten im schlimmsten Fall werden?
7. *Arithmetik I.* In ProL können positive ganze Zahlen als *Nachfolger von 0* definiert werden:

$$\begin{aligned} 0 &\equiv 0 \\ 1 &\equiv \text{succ}(0) \\ 2 &\equiv \text{succ}(\text{succ}(0)) \\ 3 &\equiv \text{succ}(\text{succ}(\text{succ}(0))) \\ &\dots \end{aligned}$$

- Implementieren Sie Prädikate *greater/2*, *add/3* und *mul/3* für die entsprechenden mathematischen Funktionen.

Z.B. gilt:

$$\text{greater}(\text{succ}(\text{succ}(\text{succ}(0))), \text{succ}(0))$$

und auch:

$$\text{add}(\text{succ}(\text{succ}(0)), \text{succ}(0), \text{succ}(\text{succ}(\text{succ}(0))))$$

- Übersetzen Sie die Klauseln des Prädikats *add/3* in WiM-Code.

8. *Wer ist ein Lügner?*. Gegeben sei folgendes ProL-Programm:

```

istLügner(X, Y)      ⇐ X = wahr, Y = lügt
istLügner(X, Y)      ⇐ X = lügt, Y = wahr
beideLügen(X, Y, Z) ⇐ X = wahr, Y = lügt, Z = lügt
beideLügen(X, Y, Z) ⇐ X = lügt, Y = lügt, Z = wahr
beideLügen(X, Y, Z) ⇐ X = lügt, Y = wahr, Z = lügt
beideLügen(X, Y, Z) ⇐ X = lügt, Y = wahr, Z = wahr
findeLügner(P, T, F) ⇐ istLügner(P, T),
                        istLügner(T, F),
                        beideLügen(F, P, T)
                        ⇐ findeLügner(Peter, Thomas, Frank)

```

Das Literal `istLügner(X, Y)` formalisiert: *X sagt, Y ist ein Lügner!*

Das Literal `beideLügen(X, Y, Z)` formalisiert: *X sagt, Y und Z lügen!*

Die Klausel für `findeLügner(F, P, T)` formalisiert die Situation, in der *P sagt, T lüge; T sagt, F lüge, und F sagt, P und T lügen.*

- Übersetzen Sie das Programm (ohne Benutzung von Indizierung)!
- Führen Sie das übersetzte Programm aus und beantworten Sie die Frage:
Wer lügt hier wirklich?
- Wie sehen die Try-Ketten für `beideLügen/3` bei Indizierung der Klauseln aus?

9. *Übersetzung ganzer Programme.* Übersetzen Sie das Programm

```

rev(X, Y, Z)  ⇐ X = [], Y = Z
rev(X, Y, Z)  ⇐ X = [H|L], rev(L, [H|Y], Z)
reverse(X, Y) ⇐ rev(X, [], Y)
               ⇐ reverse(X, [4, 2, 1])

```

in WiM-Code! Wenden Sie die Optimierung letzter Aufrufe an!

10. *Arithmetik II.* Die arithmetischen Operatoren $+$, $-$, $*$ und $/$ werden in Prolog als gewöhnliche Term-Konstrukturen aufgefasst. Um trotzdem Arithmetik durchführen zu können, erweitern wir ProL um das Prädikat `is/2`: das Ziel: `X is t` wertet den Term *t* arithmetisch aus und unifiziert das Ergebnis der Auswertung mit *X*.

- a) Erweitern Sie die WiM-Halde um den Datentyp **int** und die virtuellen Befehle für die Unifikation von *int*-Werten!
- b) Definieren Sie einen virtuellen Befehl **eval**, der den Term *t*, auf den die oberste Kellerzelle zeigt, arithmetisch auswertet und mit dem Ergebnis den Verweis oben auf dem Keller überschreibt! Wenn *t* ungebundene Variablen, Atome oder nicht-arithmetische Konstrukturen enthält, soll **eval** einen Fehler ausgeben und fehlschlagen.

- c) Definieren Sie, analog zur MaMa, virtuelle Befehle zum Durchführen arithmetischer Operationen auf dem Keller, und geben Sie ein Übersetzungsschema code_I für arithmetische Ausdrücke auf der rechten Seite eines *is*-Ziels an!
- d) Geben Sie ein Übersetzungsschema für Ziele der Form $X \text{ is } t$ an!
- e) Erzeugen Sie mithilfe des neuen Schemas Code für das Prädikat:

$$\text{len}(X, L) \Leftarrow X = [], L \text{ is } 6 - 2 \cdot 3$$

$$\text{len}(X, L) \Leftarrow X = [_|R], \text{len}(R, L'), L \text{ is } L' + 1$$

4.16 Liste der Register der WiM

BP , Rücksetzzeiger (Backtrack Pointer)	S. 132
FP , Rahmenzeiger (Frame Pointer)	S. 113
PC , Befehlszähler (Program Counter)	S. 112
HP , Haldenzeiger (Heap Pointer)	S. 113
SP , Kellerzeiger (Stack Pointer)	S. 113
TP , Spurzeiger (Trail Pointer)	S. 132

4.17 Liste der Codeerzeugungsfunktionen der WiM

code_A	S. 114	code_C	S. 130
code_G	S. 118	code_P	S. 131
code_U	S. 125	code	S. 137

4.18 Liste der WiM-Instruktionen

bind	S. 121	no	S. 139	setcut	S. 148
call	S. 119	pop	S. 126	slide	S. 142
check	S. 127	popenv	S. 137	son	S. 129
delbtp	S. 135	prune	S. 147	trim	S. 143
fail	S. 145	pushenv	S. 131	try	S. 135
getNode	S. 146	putanon	S. 116	uatom	S. 125
halt	S. 139	putatom	S. 116	unify	S. 122
init	S. 139	putref	S. 116	up	S. 129
index	S. 146	putvar	S. 116	uref	S. 126
jump	S. 142	putstruct	S. 116	uvar	S. 126
lastcall	S. 141	setbtp	S. 135	ustruct	S. 127
lastmark	S. 141				

4.19 Literaturhinweise

Einführende Bücher in die Programmierung mit Prolog sind [CM03], [Han86] und [KBS86]. [SS94] und [MW88] enthalten neben einer Darstellung von Programmiermethoden auch Abschnitte über die Interpretation bzw. Übersetzung von Prolog-Programmen. Die Grundlagen der logischen Programmierung sind z.B. in [Llo87], [Apt90] und [Bez88] dargestellt.

David H.D. Warren beschreibt in [War77] eine erste virtuelle Maschine für Prolog. Diese benutzt statt *structure copying* noch *structure sharing*. Die WAM (Warren Abstract Machine), die Grundlage der meisten verfügbaren Prolog-Implementierungen, wurde von ihm in [War83] definiert. Sie wird in [AK91] didaktisch geschickt schrittweise erläutert.

Weiterentwicklungen der virtuellen Maschinen für Prolog bilden die Basis moderner, auf Prolog basierender Programmiersysteme wie SWI-Prolog [SWD05], SICStus-Prolog [SIC06] und Eclipse [AW06]. Sie bilden auch den Ausgangspunkt zur Implementierung funktional-logischer Sprachen wie Curry [HS99].

Die Übersetzung objektorientierter Programmiersprachen

Softwaresysteme werden zunehmend komplexer und größer. Damit wächst die Notwendigkeit, die Entwicklung solcher Systeme effizienter und transparenter zu machen. Eine Hoffnung besteht darin, Softwaresysteme – wie heute schon Hardware-systeme (und die allermeisten Produkte des täglichen Lebens, z.B. Autos, Waschmaschinen usw.) – aus vorgefertigten Standardbausteinen zusammenzusetzen. Dieser Hoffnung versucht man u.a. durch folgende Ideen näherzukommen:

- Modularisierung,
- Wiederverwendbarkeit von Modulen,
- Erweiterbarkeit von Modulen, und
- Abstraktion.

Objektorientierte Sprachen bieten neue Möglichkeiten in diesen Bereichen. Objektorientierung wird deshalb heute als ein wesentliches Paradigma angesehen, um die Komplexität von Softwaresystemen zu beherrschen. In diesem Kapitel skizzieren wir die wichtigsten Konzepte objektorientierter Sprachen.

5.1 Konzepte objektorientierter Sprachen

Objektorientierte Sprachen sind imperativen Sprachen weit näher verwandt als funktionale oder gar logische Programmiersprachen. Sie benutzen (typischerweise) dasselbe Ausführungsmodell: eine Variante des von-Neumann-Rechners; unter expliziter Programmkontrolle wird ein komplexer Zustand verändert. Man kann objektorientierte Sprachen deshalb als imperative Sprachen ansehen, zu deren Konzepten wie Variablen, Felder, Strukturen, Prozeduren und Funktionen weitere Konzepte hinzukommen. Im Zentrum stehen Abstraktionsmechanismen und zugehörige Modularisierungsmöglichkeiten.

5.1.1 Objekte

Wie wir gesehen haben, bieten imperative Programmiersprachen funktionale Abstraktion an; eine eventuell komplexe Berechnung kann in eine Funktion (Proze-

dur) „verpackt“ werden und durch Aufrufe aktiviert werden. Deshalb ist die Hauptmodularisierungseinheit imperativer Sprachen die Funktion bzw. Prozedur. Solange die Komplexität der Daten gegenüber der der Abläufe vernachlässigbar ist, ist dies eine geeignete Abstraktions- und Modularisierungsstufe. Für Aufgaben, deren Beschreibung und effiziente Lösung die Verwendung komplexer Datenstrukturen erfordert, sind Funktionen allein als Modularisierungseinheit möglicherweise nicht ausreichend. Eine angemessenes Abstraktionskonzept für die effiziente Entwicklung solcher Programme sollte es erlauben, sowohl die Datenstrukturen als auch die zugehörigen auf diesen Strukturen arbeitenden Funktionen in einer Einheit zu kapseln. Dies nennt man *Datenabstraktion*.

Das Grundkonzept objektorientierter Sprachen ist das *Objekt*. Ein Objekt besteht aus einem Objektzustand, ausgedrückt durch die aktuellen Werte eines Satzes von *Attributen*, und Funktionen, die auf diesem Zustand arbeiten, den *Objekt-Methoden*. Attribute und Methoden eines Objektes zusammen nennen wir seine *Merkmale*. Ein Objekt kapselt in seinem Zustand sowohl Daten als auch in seinen Methoden die auf diesen Daten durchführbaren Operationen. Die wichtigste Basisoperation objektorientierter Sprachen ist die Aktivierung einer Methode f für ein Objekt o , etwa geschrieben als $o.f$. Im Vordergrund steht dabei das Objekt; die Methode ist diesem als sein Bestandteil untergeordnet. Diese Objektorientierung hat der Sprachklasse ihren Namen gegeben.

5.1.2 Objektklassen

Um die Programmentwicklung sicherer und effizienter zu machen, ist es wünschenswert, dass Inkonsistenzen und Fehler in Programmen möglichst frühzeitig und möglichst zuverlässig erkannt werden. Übersetzer können hierzu einen Beitrag leisten, da sie Programmteile zum Teil eingehend analysieren. Diese Prüfung auf Konsistenz und Fehlerfreiheit ist dabei notwendigerweise nicht vollständig, da sie nur mit partiellen Spezifikationen arbeitet. Typinformation spielt dabei eine wesentliche Rolle.

(Statische) Typinformation besteht aus Angaben zu den in einem Programm verwendeten Namen. Sie legen fest, dass die an diese Namen gebundenen Laufzeitobjekte einen angegebenen Typ besitzen. Ein Typ steht dabei für eine Menge zulässiger Werte; er legt fest, welche Operationen auf die an den Namen gebundenen oder ihnen zugewiesenen Werte angewendet werden dürfen. Übersetzer können Typinformation nutzen, um

- Inkonsistenzen zu erkennen,
- Mehrdeutigkeiten in der Operatorverwendung aufzulösen,
- automatische Typkonversionen einzufügen und
- effizienteren Code zu erzeugen.

Statische, das heißt dem Übersetzer bekannte oder von ihm ableitbare Typinformation ist eine wichtige Voraussetzung dafür, dass er effiziente und einigermaßen zuverlässige Programme erzeugen kann.

Nicht alle objektorientierten Sprachen benutzen statische Typisierung, z.B. nicht Smalltalk-80. Sprachen wie C++, Java oder C# dagegen erweitern die von imperativen Sprachen wie etwa Pascal oder C bekannten Typkonzepte. Ihre Typen heißen

üblicherweise *Objektklassen* oder kurz: *Klassen*. Eine Objektklasse legt Attribute und Methoden fest, die ein Objekt haben muss, um zu dieser Klasse zu gehören. Für Attribute wird ihr Typ und für Methoden ihr Prototyp (Typen für Rückgabewert und Parameter) vorgegeben. Einige objektorientierte Sprachen, so etwa Eiffel, erlauben weitere Festlegungen für die Methoden, z.B. Vor- und Nachbedingungen. Auf diese Weise kann die Bedeutung der Methode (Semantik) genauer spezifiziert werden. Häufig definiert die Klasse auch die Methoden; diese Definitionen können jedoch unter gewissen Bedingungen überschrieben werden. Objekte, die neben den geforderten weitere Merkmale besitzen, können u.U. ebenfalls der Klasse angehören.

Die Objektklasse ist das Konzept zur Datenabstraktion in objektorientierten Sprachen. Sie fungiert als *Generator* für Objekte, ihre *Instanzen*.

5.1.3 Vererbung

Unter *Vererbung*, eigentlich *Beerbung*, (engl. Inheritance) versteht man die Übernahme aller Merkmale einer Klasse *B* in eine neue Klasse *A*. Die Klasse *A* kann zusätzlich Merkmale definieren und unter gewissen Voraussetzungen von *B* geerbte Methoden überschreiben.

Erbt *A* von *B*, dann heißt *A* eine von *B* *abgeleitete Klasse* oder auch *Unterklasse* von *B*; *B* heißt *Basisklasse* oder auch *Oberklasse* von *A*.

Die Vererbung gehört zu den wichtigsten Konzepten objektorientierter Sprachen. Sie vereinfacht Erweiterungen und Variantenbildung entscheidend. Durch Bildung von Vererbungshierarchien erlaubt sie ferner, Klassenbibliotheken zu strukturieren und verschiedene Abstraktionsstufen einzuziehen.

Das Vererbungskonzept verschafft uns eine Möglichkeit, in einfacher Weise auf Teile einer bestehenden Implementierung zurückzugreifen, sie zu erweitern und sie bei Bedarf lokal – durch Überschreiben einzelner Methoden – an spezielle Anforderungen oder Gegebenheiten anzupassen. Ferner erhalten wir die Möglichkeit, *abstrakte* Klassen zu definieren. Abstrakte Klassen sind Klassen, die undefinierte Methoden enthalten. Sie besitzen keine Instanzen, das heißt Objekte, die nicht aus einer echten Unterklasse stammen. Dies führt eine ähnliche Flexibilität in Programmiersprachen ein, wie wir sie in der natürlichen Sprache durch abstrakte Begriffe und unterschiedliche Abstraktionsstufen gewinnen. Was auf einer höheren Abstraktionsstufe formuliert werden kann, hat einen weiteren Anwendungsbereich und damit ein höheres Maß an Wiederverwendbarkeit.

Typisierte objektorientierte Sprachen besitzen deshalb eine Vererbungshierarchie in ihrem Typsystem. Erbt eine Klasse *A* öffentlich von der Klasse *B*, dann ist der *A* zugeordnete Typ ein *Teiltyp* des Typs zu *B*. Jedes Objekt eines Teiltyps ist automatisch auch Element des Obertyps; eine erbende Klasse wird Teilklass der beerbten Klasse. Dies hat zur Folge, dass an einer Eingabeposition (Funktionsparameter, rechte Seiten von Zuweisungen) oder als Funktionsrückgabewert Objekte eines beliebigen Teiltyps des angegebenen Typs auftreten dürfen. Dies nennen wir auch die *Teiltypregel*.

Eine Einschränkung dieses nützlichen Prinzips wollen wir aber nicht verschweigen: Objekte der Unterklassen von *B* verfügen eventuell über zusätzliche Merkmale

und können deshalb nicht leicht in dem Platz abgelegt werden, der für *B*-Objekte reserviert werden müsste. Dies ist natürlich etwas anderes, wenn die Programmiersprache in Wahrheit nicht mit Objekten selbst, sondern mit Verweisen auf Objekten hantiert wie etwa die Programmiersprachen Eiffel oder Java: Verweise beanspruchen stets die gleiche Speichergröße — unabhängig von der Klasse des Objekts, auf das sie zeigen. Hier ist die Teiltypregel in vollem Umfang anwendbar. Im Gegensatz dazu unterscheidet die Programmiersprache C++ genau zwischen Objekten und Verweisen auf Objekten. C++ ruft deshalb für einen Parameter, dessen Typ eine Klasse *A* ist, gegebenenfalls den *copy*-Konstruktor *A*(**const** *A*& *x*) der Klasse *A* auf. Dieser Konstruktor soll für eine angemessene *Typanpassung* sorgen.

Wir nennen die Objekte von *A*, die nicht auch Objekte einer echten Teilklassse sind, die *eigentlichen Objekte* von *A*. Entsprechend nennen wir *A* den *eigentlichen Typ* der eigentlichen *A*-Objekte. Damit besitzt jedes Objekt einen eindeutig bestimmten eigentlichen Typ: den kleinsten Typ, zu dem das Objekt gehört. Es ist darüber hinaus Element jedes Obertyps seines eigentlichen Typs.

Aufgrund der Teiltypregel können Methoden und Funktionen in objektorientierten Sprachen auf einer Parameterposition (Verweise auf) Objekte mit verschiedenen eigentlichen Typen und damit unterschiedlichem Aufbau akzeptieren. Dies ist eine Form von *Polymorphie*.

Die Teiltypregel der Vererbung zusammen mit der Möglichkeit, dass erbende Klassen eine geerbte Methode überschreiben dürfen, hat eine interessante und für Übersetzer wichtige Konsequenz, die tatsächlich bei einem Methodenaufruf *e.f*(...) aufgerufene Methode *f* muss sich nach dem eigentlichen Typ des Objekts richten, zu dem sich der Ausdruck *e* zur *Laufzeit* auswertet. Damit muss der Übersetzer Code für die Aktivierung einer Methode erzeugen, die er zu diesem Zeitpunkt u.U. noch nicht kennt. Offenbar haben wir es mit einer Form von *dynamischer Bindung* zu tun. Der Hauptteil dieses Kapitels wird sich mit einer effizienten Implementierung von Methodenaufrufen in Anwesenheit von Vererbung beschäftigen.

5.1.4 Generizität

Streng typisierte Sprachen zwingen häufig zu einer Reimplementierung der gleichen Funktion für verschiedene Typen. Oft unterscheiden sich die Funktionen ausschließlich im Typ ihrer Parameter. Diese mehrfachen Funktionsinstanzen erschweren die Implementierung, machen Programme unübersichtlicher und schwerer wartbar.

Wir haben gesehen, dass das auf Vererbung beruhende Typkonzept objektorientierter Sprachen uns in einigen Fällen eine Duplikation von Funktionsimplementierungen erspart. Für eine wichtige Klasse von Problemstellungen führt Vererbung allein aber nicht zu eleganten Lösungen. Die Implementierung allgemeiner Behälterdatenstrukturen wie Listen, Keller oder Wartenschlangen haben einen natürlichen Parameter: den Typ ihres Inhalts. *Generizität* (engl. Genericity) ermöglicht uns, für solche Datenstrukturen und ihre Methoden eine Mehrfachimplementierung zu vermeiden. Sie erlaubt, Typdefinitionen (und Funktionsdefinitionen) zu parametrisieren; als Parameter sind selbst wieder (möglicherweise eingeschränkte) Typen zugelassen.

Eine einzige Definition für die parametrisierte Klasse $\text{list}\langle t \rangle$ kann beispielsweise Listen mit beliebigem Elementtyp t beschreiben. Listen mit spezifischem Typ werden durch *Instantiierung* der generischen Klasse erzeugt: $\text{list}\langle \text{int} \rangle$ bezeichnet etwa eine Liste, deren Elemente vom Typ **int** sind.

Generizität wird von einigen objektorientierten Sprachen wie z.B. C++ und Java unterstützt. Sie ist allerdings keine Erfindung objektorientierter Sprachen. Generizität ist auch ein wesentliches Merkmal moderner funktionaler Programmiersprachen wie etwa OCaml oder Haskell. Aber auch die imperative Sprache Ada hatte bereits früher ein sehr ausgefeiltes Konzept von Generizität.

5.1.5 Informationskapselung

Die meisten objektorientierten Sprachen stellen Konstrukte zur Verfügung, mit denen die Merkmale einer Klasse als *privat* oder *öffentlich* klassifiziert werden können. Private Merkmale sind in gewissen Kontexten entweder ganz unsichtbar oder zumindest nicht zugreifbar. Manche objektorientierten Sprachen unterscheiden verschiedene Sichtbarkeitskontexte, etwa innerhalb der Klasse, in abgeleiteten Klassen, in fremden Klassen, in bestimmten Klassen usw. Sprachkonstrukte oder allgemeine Regeln können festlegen, in welchen Kontexten welche Merkmale sichtbar bzw. lesbar/schreibbar oder aufrufbar sind.

Die Realisierung solcher Konstrukte durch einen Übersetzer ist einfach und naheliegend. Wir werden daher in diesem Kapitel nicht weiter darauf eingehen, obwohl die Informationskapselung von großer Bedeutung ist für eine klare Trennung zwischen der abstrakten Sicht der Klassenbedeutung und der konkreten Sicht ihrer Implementierung.

5.1.6 Zusammenfassung

Wir fassen die wichtigsten Ergebnisse unserer bisherigen Diskussion zusammen:

- Objektorientierte Sprachen bieten als weitere Modularisierungseinheit *Objektklassen*. Objektklassen können sowohl Daten als auch auf diesen Daten arbeitende Funktionen kapseln.
- Das *Vererbungskonzept* ist ein Hilfsmittel zur Erweiterung und Variantenbildung bestehender Objektklassen.
- Das Typsystem objektorientierter Sprachen nutzt das Vererbungskonzept: Erben- de Klassen werden Teiltypen der Basisklassen; ihre Objekte können an (fast) allen Stellen benutzt werden, an denen Objekte der Basisklasse zulässig sind.
- *Vererbungshierarchien* führen unterschiedliche Abstraktionsebenen in Programme ein. Dies erlaubt, an verschiedenen Stellen innerhalb eines Programms oder Systems auf unterschiedlichen Abstraktionsstufen zu arbeiten.
- *Abstrakte* Klassen können in Spezifikationen verwendet und durch schrittweise Vererbung verfeinert und schließlich realisiert werden. Der Übergang zwischen Spezifikation, Entwurf und Realisierung wird so fließend.

- *Generizität* erlaubt, Klassendefinitionen zu parametrisieren. So können etwa allgemeine Algorithmen und zugehörige Datenstrukturen wie Listen, Stacks, Warteschlangen, Mengen... unabhängig vom Elementdatentyp implementiert werden.

Beispiele für objektorientierte Sprachen sind C++, Eiffel, sowie Java und C#. In den letzten Jahren hat gerade die objektorientierte Sprache Java immens an Bedeutung gewonnen – unter anderem als Basis für plattformunabhängige Software. Das Sandkasten-Prinzip, ursprünglich entwickelt, um über das Internet geladenen Code sicher ausführen zu können, bewährt sich nun auch im Serverbereich.

Als Urvater objektorientierter Sprachen gilt Simula 67, eine Erweiterung von Algol 60 um Objektklassen, einfache Vererbung, Coroutinen und Primitive zur Unterstützung diskreter Simulationen. Ein weiterer bekannter Vertreter ist Smalltalk-80, eine normalerweise interpretierte Sprache ohne statische Typisierung, mit einfacher Beerbung, in der Klassen selbst wieder Objekte sind, die unter Programmkontrolle verändert werden können. Weitere Vertreter sind Objective-C, eine Erweiterung von C um Smalltalk-80-Konzepte, und die objektorientierten Erweiterungen von Lisp wie Loops oder Flavors.

5.2 Eine objektorientierte Erweiterung von C

Als konkretes Beispiel für eine objektorientierte Sprache wählen wir eine einfache Teilmenge von C++. Klassen werden aufgefasst als Erweiterungen von Verbunden. Sie enthalten *Attribute*, das sind Datenfelder, und *Methoden*, die gegebenenfalls als **virtual** deklariert sind und damit überschrieben werden können, sowie Konstruktoren, die bei Anlegen von Objekten der Klasse für ihre Initialisierung sorgen.

Beispiel 5.2.1 Betrachten Sie die folgende Definition einer Klasse list:

```

int count ← 0;
class list {
    int info;
    list * next;
    list (int x) {
        info ← x; count++; next ← null;
    }
    virtual int last () {
        if (next = null) return info;
        else return next → last ();
    }
};

```

Die Klasse verfügt über die beiden Attribute *info* und *next* der Typen **int** bzw. list *. Darüber hinaus gibt es einen Konstruktor, um neue Listenobjekte herzustellen sowie

eine Methode *last*, die den Inhalt des Attributs *info* des letzten über *next*-Verweise erreichbaren *list*-Objekts zurückliefert. Weil diese Methode als **virtual** gekennzeichnet ist, darf sie gegebenenfalls in Unterklassen neu definiert werden.

Gemäß der Philosophie von C++ müssen Objekte nicht notwendigerweise in der Halde liegen, sondern können wie Verbunde auch direkt auf dem Keller allokiert werden. Dies ist ein wesentlicher Unterschied zu Java. Objekte in Java würden in C++ als *Verweise* auf Halden-allokierte C++-Objekte aufgefasst werden.

Beachten Sie, dass wir hier zur Vereinfachung auf jegliche Angaben zur Sichtbarkeit einzelner Attribute oder Methoden verzichtet haben. Um ein lauffähiges C++-Programm zu erhalten, könnten wir deshalb z.B. sämtliche Komponenten der Klasse als **public** deklarieren. □

Im Folgenden wollen wir uns ansehen, wie unsere Implementierung von C auf Klassen, Objekte und ihre Komponenten erweitert werden kann.

5.3 Die Speicherorganisation für Objekte

Die Idee zur Implementierung von Objekten besteht darin, nur diejenigen Dinge im Objekt selbst anzulegen, die sich von Objekt zu Objekt unterscheiden. Methoden, die nicht als **virtual** deklariert sind, ergeben sich direkt aus dem Typ des Objekts und brauchen damit nicht abgespeichert werden. Attribute und als **virtual** deklarierte Methoden können dagegen nicht immer zur Übersetzungszeit ermittelt werden. Damit die einzelnen Attribute einer Klasse in allen Unterklassen gleich adressiert werden können, werden wir bei der Implementierung darauf achten, dass die jeweiligen neuen Attribute einer Unterklasse stets hinten an die Attribute der Oberklasse angefügt werden. Für die virtuellen Methoden beobachten wir, dass sie, d.h. ihre Anfangsadressen, sich alleine aus dem *Laufzeittyp* des Objekts ermitteln lassen. Ihre Anfangsadressen werden deshalb nicht in jedem Objekt selbst abgespeichert. Stattdessen legen wir für jede Klasse *A* vor der Programmausführung eine Tabelle t_A an, in der die Anfangsadressen der virtuellen Methoden für alle eigentlichen *A*-Objekte abgelegt sind. In einem *A*-Objekt selbst brauchen wir dann nur noch einen Verweis auf die Tabelle t_A unterzubringen.

Beispiel 5.3.1 Betrachten Sie ein weiteres Mal unsere Definition der Klasse *list* aus Beispiel 5.2.1. Die Speicherorganisation eines Objekts der Klasse *list* enthält einen Verweis auf die Tabelle für die virtuellen Methoden der Klasse *list*, gefolgt von den beiden Zellen für die Attribute *info* und *next* (Abb. 5.1).

Nehmen wir an, die Unterklasse *mylist* sei definiert durch:

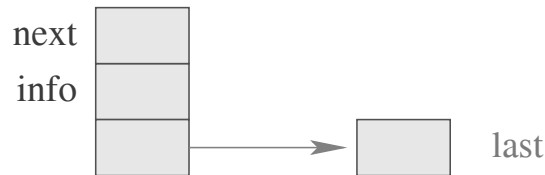


Abb. 5.1. Ein Objekt der Klasse list.

```

class mylist : list {
    int moreInfo;
    virtual int length () {
        if (next == null) return 1;
        else return 1 + next → length ();
    }
};

```

Beachten Sie, dass wir hier erneut auf Sichtbarkeitsangaben verzichtet haben. In C++ müsste man etwa eine Qualifizierung **public** vor der Oberklasse einfügen. Andernfalls könnten die Attribute und Methoden der Oberklasse nur innerhalb der erbbenden Klasse verwendet werden. Objekte der Klasse **mylist** enthalten an der Relativadresse 0 einen Verweis auf die Tabelle der virtuellen Funktionen der Klasse **mylist**. Diese unterscheidet sich von der Tabelle der Oberklasse dadurch, dass zusätzlich die Methode **length** vermerkt ist. Ansonsten wird hinter den beiden Feldern für die Attribute der Oberklasse **list** ein weiteres Feld für das zusätzliche Attribut *moreInfo* bereitgestellt (Abb. 5.2). □

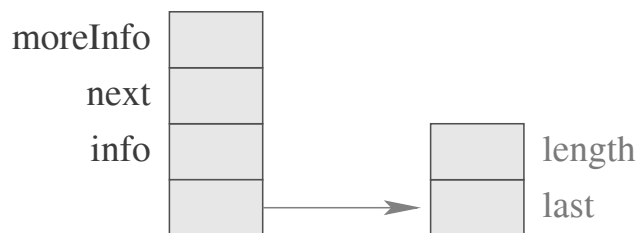


Abb. 5.2. Ein Objekt der Unterklasse mylist.

Für die Übersetzung nehmen wir an, wir hätten für jede Klasse *A* eine Adressumgebung ρ_A . Die Adressumgebung ρ_A bildet jeden in der Klasse *A* sichtbaren Namen *x* auf eine erweiterte Relativadresse ab. Wir unterscheiden die folgenden Arten von Namen:

globale Variable	(G, a)
Attribut	(A, a)
überschreibbare Methode	(V, a)
nicht überschreibbare Methode	(N, a)

Beim Aufbau der Adressumgebungen innerhalb von Methoden oder Konstruktoren können zu einer solchen Adressumgebung gegebenenfalls noch formale Parameter und lokale Variablen hinzukommen, welche wie bei der Übersetzung von C das Etikett L erhalten. Bei nicht überschreibbaren Methoden x enthält $\rho_A(x)$ die Anfangsadresse des Codes für x . Bei überschreibbaren Methoden x dagegen liefert $\rho_A(x)$ nicht die Anfangsadresse von x , sondern die Relativadresse innerhalb von Objekten der Klasse A , an der die Anfangsadresse von x abgelegt ist.

Für die verschiedenen Arten von Variablen wird Code zur Berechnung des L-Werts wie folgt erzeugt:

$$\text{code}_L x \rho = \begin{cases} \text{loadc } a & \text{falls } \rho(x) = (G, a) \\ \text{loadr } a & \text{falls } \rho(x) = (L, a) \\ \text{loadr } -3 \\ \text{loadc } a \\ \text{add} & \text{falls } \rho(x) = (A, a) \end{cases}$$

Die Attribute des aktuellen Objekts werden relativ zu dem Verweis **this** auf dieses Objekt adressiert. Diesen Verweis **this** finden wir auf dem Keller an der festen Adresse -3 relativ zum gegenwärtigen Rahmenzeiger FP . Wie Sie sicherlich schnell nachgerechnet haben, entspricht diese Festlegung einer Übergabe dieses Verweises an die Methode als erstes Argument.

Anstelle des Abspeicherns von **this** im aktuellen Kellerrahmen hätten wir auch ein neues Register *COP* (*Current Object Pointer*) einführen können. Dieses müsste dann jeweils vor Aufrufen von nicht-überschreibbaren Methoden gerettet und nach Rückkehr aus solchen Aufrufen restauriert werden. Wir haben davon abgesehen, um die Architektur der C-Maschine so weit wie möglich weiterzuverwenden.

Zur Optimierung der Adressierung von Objektattributen können wir etwa die folgenden abgeleiteten Befehle einführen:

$$\begin{aligned} \text{loadmc } q &= \text{loadr } -3 \\ &\quad \text{loadc } q \\ &\quad \text{add} \end{aligned}$$

$$\begin{aligned} \text{loadm } q \ m &= \text{loadmc } q \\ &\quad \text{load } m \end{aligned}$$

$$\begin{aligned} \text{storem } q \ m &= \text{loadmc } q \\ &\quad \text{store } m \end{aligned}$$

wobei wir bei **loadm** und **storem** wieder auf das zweite Argument verzichten, sofern es 1 ist. Diese Befehle ließen sich natürlich in Anwesenheit eines Registers *COP* besonders effizient implementieren.

5.4 Methodenaufrufe

Ein Methodenaufruf ist von der Form:

$$e_1.f(e_2, \dots, e_n)$$

Einen Aufruf $f(e_2, \dots, e_n)$ ohne Angabe des Objekts fassen wir dabei als eine Abkürzung auf für den Aufruf:

$$\mathbf{this} \rightarrow f(e_2, \dots, e_n) \quad \text{bzw.} \quad (*\mathbf{this}).f(e_2, \dots, e_n)$$

Einen Aufruf $e_1.f(e_2, \dots, e_n)$ relativ zu dem Objekt e_1 behandeln wir wie einen normalen Funktionsaufruf, bei dem als erstes Argument ein Verweis auf das Objekt o übergeben wird, zu dem sich e_1 auswertet. Im Falle einer überschreibbaren Methode f müssen wir nur darauf achten, dass ein Aufruf von f indirekt durch das im Objekt o für f vorgesehene Feld geschieht.

Um die Darstellung der Übersetzungsschemata zu vereinfachen, betrachten wir im Folgenden nur nicht-variable Argumentlisten und nehmen zudem an, dass der Platz für die aktuellen Parameter (einschließlich des übergebenen Objektverweises) für die Abspeicherung eines eventuellen Rückgabewerts ausreicht. Dann erhalten wir für eine nicht-überschreibbare Methode:

$$\begin{aligned} \text{code}_R \ e_1.f(e_2, \dots, e_n) \ \rho &= \text{code}_R \ e_n \ \rho \\ &\quad \dots \\ &\quad \text{code}_R \ e_2 \ \rho \\ &\quad \text{code}_L \ e_1 \ \rho \\ &\quad \mathbf{mark} \\ &\quad \mathbf{loadc} \ _f \\ &\quad \mathbf{call} \end{aligned}$$

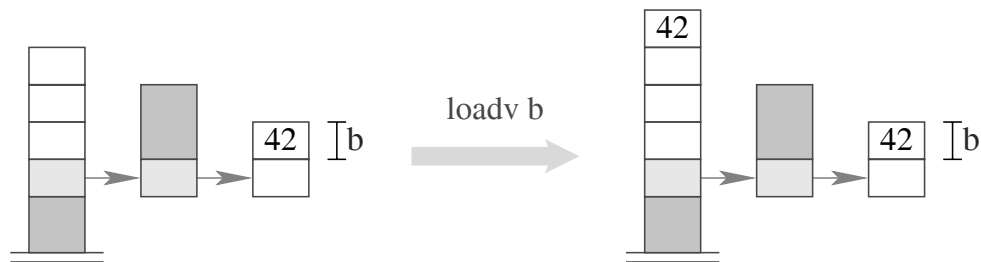
sofern $\rho_A(f) = (N, _f)$ für die statisch bekannte Klasse A des Ausdrucks e_1 ist.

Beachten Sie, dass das Objekt, zu dem sich e_1 auswertet, *by reference* übergeben wird. Technisch bedeutet das, dass für den ersten Parameter e_1 Code zur Berechnung des L-Werts erzeugt wird und nicht Code zur Berechnung des R-Werts wie für die übrigen Parameter.

Die Übersetzung des Aufrufs einer überschreibbaren Methode unterscheidet sich von der Übersetzung einer nicht überschreibbaren Methode einzig darin, dass die anzuspriechende Codeadresse erst zur Laufzeit ermittelt werden kann:

$$\begin{aligned} \text{code}_R \ e_1.f(e_2, \dots, e_n) \ \rho &= \text{code}_R \ e_n \ \rho \\ &\dots \\ &\text{code}_R \ e_2 \ \rho \\ &\text{code}_L \ e_1 \ \rho \\ &\mathbf{mark} \\ &\mathbf{loadv} \ b \\ &\mathbf{call} \end{aligned}$$

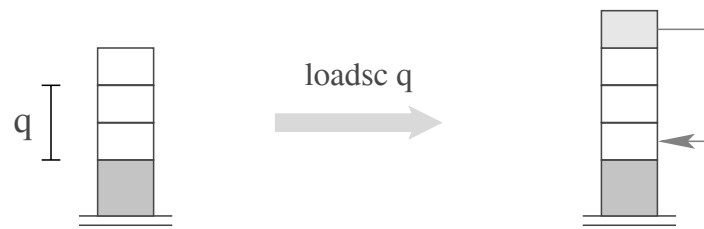
sofern für die statische Klasse A des Ausdrucks e_1 $\rho_A(f) = (V, b)$ ist. Die neue Instruktion **loadv** b berechnet aus der Relativadresse b von f innerhalb des Objekts o , zu dem sich e_1 auswertet, die tatsächliche Anfangsadresse der aktuell gemeinten Implementierung von f (Abb. 5.3). Die Anfangsadresse a des Objekts o wird als erster Parameter übergeben. Sie steht deshalb im Keller unterhalb der organisatorischen Zellen an der Adresse $SP - 3$. Mit ihr wird die Anfangsadresse der aktuellen Tabelle t der virtuellen Methoden des Objekts o ermittelt. Innerhalb dieser Tabelle findet sich die Anfangsadresse der Methode an der Adresse b .



$$S[SP + 1] \leftarrow S[S[S[SP - 3]] + b]; \ SP++;$$

Abb. 5.3. Die Instruktion **loadv** b .

Beachten Sie, dass die Instruktion **loadv** b auf das Objekt o relativ zum SP zugreift. Generell können wir solche relativen Zugriffe implementieren, wenn wir eine Instruktion **loadsc** q bereitstellen, die die negierte Konstante q relativ zum Kellerpegel SP lädt (Abb. 5.4).



$$S[SP + 1] \leftarrow SP - q; SP++;$$

Abb. 5.4. Die Instruktion **loadsc** q .

Wie bei dem Laden (und Speichern) relativ zum Rahmenzeiger FP erlaubt diese Instruktion die Implementierung einer Instruktion **loads** q , die nicht die Adresse, sondern den Inhalt der Speicherzelle $-q$ relativ zum SP lädt:

loads q = **loadsc** q
load

Die Instruktion **loadv** b könnten wir dann auch realisieren durch die Folge:

loadv b = **loads** 3
load
loadc b
add
load

Beispiel 5.4.1 Betrachten wir erneut die Klasse `list` aus Beispiel 5.2.1. Die Adressumgebung für diese Klasse ist gegeben durch:

$$\rho_{\text{list}} = \{info \mapsto (A, 1), next \mapsto (A, 2), last \mapsto (V, 0)\}$$

Der rekursive Aufruf $next \rightarrow last()$ im Rumpf der überschreibbaren Methode `last` wird damit übersetzt in die Folge:

loadm 2
mark
loadv 0
call

□

5.5 Die Definition von Methoden

Im Allgemeinen wird die Definition einer Methode f einer Klasse A von der folgenden Form sein:

$$t \ f(t_2 \ x_2, \dots, t_n \ x_n) \ \{ \text{ss} \}$$

Für die Implementierung fassen wir solche Methoden als Funktionen mit einem zusätzlichen ersten Argument auf, welches den Verweis auf das aktuelle Objekt o enthält. Innerhalb des Rumpfs ss der Methode bezeichnet das Schlüsselwort **this** diesen Verweis. Entsprechend der Übergabe als erstem Argument steht dieser Verweis im Kellerrahmen an der Relativadresse -3 . Deshalb übersetzen wir:

$$\text{code}_R \text{ this } \rho = \text{loadr } -3$$

Insgesamt ändert sich am Codeschema für die Übersetzung von Methoden in C++ gegenüber der Übersetzung von Funktionen in C nichts.

Beispiel 5.5.1 Wieder betrachten wir die Klasse `list` aus Beispiel 5.2.1. Die Implementierung der Methode `last` liefert:

<code>_last :</code>	enter 6	loadm 1	loadv 0
	alloc 0	storer -3	call
	loadm 2	return	storer -3
	loadc 0		return
eq	$A :$	loadm 2	
jumpz A		mark	

Beachten Sie, dass wir an der Adresse A den Code zur Ermittlung des L-Werts des Ausdrucks $(*next)$ erzeugen, d.h. den Code zur Ermittlung des R-Werts von $next$. Dieser besteht gerade aus der Instruktion **loadm** 2.

Die InSTRUCTIONSfolge für die Methode `last` kommt uns vertraut vor. Die einzigen Unterschiede etwa zu einer entsprechenden InSTRUCTIONSfolge für C liegen darin, dass wir spezialisierte Instruktionen **loadm** und **storem** verwenden, um auf die Attribute des aktuellen Objekts zuzugreifen. Auch benötigen wir Instruktionen **loadv**, um die Anfangsadressen der überschreibbaren Methoden zu finden. \square

5.6 Die Verwendung von Konstruktoren

Objektorientierte Sprachen wie C++ bieten die Möglichkeit, für eine Klasse A Konstruktoren zu definieren, mit denen ein neu angelegtes Objekt initialisiert werden kann. Wir unterscheiden hier zwei Arten der Objekterzeugung:

- (1) direkt, wie z.B. auf der rechten Seite einer Zuweisung:

$$A(e_2, \dots, e_n)$$

Als R-Wert erhalten wir das (initialisierte) Objekt selbst.

(2) indirekt auf der Halde:

new $A(e_2, \dots, e_n)$

Der R-Wert ist hier ein Verweis auf das neu angelegte Objekt.

Wir beginnen mit der Behandlung indirekter Objekterzeugung. Von einer Implementierung erwarten wir, dass sie zuerst Platz für das neue Objekt auf der Halde allokiert. Dann sollten die Felder für die virtuellen Funktionen initialisiert werden. Den eigentlichen Konstruktoraufruf behandeln wir wieder analog zu Funktionsaufrufen in C. Wir müssen darauf achten, dass der Verweis auf das neue Objekt als erster (impliziter) Parameter an den Konstruktor übergeben und nach Beendigung des Aufrufs auch oben auf dem Keller hinterlassen wird. Dieses leistet das folgende Schema:

$$\text{code}_R(\text{new } A(e_2, \dots, e_n)) \rho = \begin{array}{l} \text{loadc } |A| \\ \text{new} \\ \text{code}_R e_n \rho \\ \dots \\ \text{code}_R e_2 \rho \\ \text{loads } m \quad // \text{ lädt relativ zum } SP \\ \text{mark} \\ \text{loadc } _A \\ \text{call} \end{array}$$

wobei m der Platzbedarf für die aktuellen Parameter e_2, \dots, e_n , $|A|$ der Platzbedarf für die Instanzen von A und $_A$ die Anfangsadresse des Codes für den aktuellen Konstruktor sind.

Betrachten wir nun die Anwendung eines Konstruktors in einem normalen Ausdruck. Hier fordert die Semantik, dass die Auswertung zu einem R-Wert das neue Objekt nicht in der Halde, sondern auf dem Keller anlegt und dort zurückliefert. In diesem Fall wird deshalb zuerst genügend viel Platz auf dem Keller allokiert, bevor die Argumente des Konstruktoraufrufs ausgewertet werden. Als erster Parameter wird schließlich an den Konstruktoraufruf einen Verweis auf das neu erzeugte Objekt übergeben. Dies ergibt das folgende Übersetzungsschema:

$$\text{code}_R(A(e_2, \dots, e_n)) \rho = \begin{array}{l} \text{alloc } |A| \\ \text{code}_R e_n \rho \\ \dots \\ \text{code}_R e_2 \rho \\ \text{loadsc } q \\ \text{mark} \\ \text{loadc } _A \\ \text{call} \end{array}$$

wobei $_A$ die Anfangsadresse des aufzurufenden Konstruktors für die Klasse A ist. Die Anfangsadresse des neu angelegten Objekts lässt sich am einfachsten relativ zum Kellerpegel ermitteln. Falls m der Platzbedarf für die aktuellen Parameter e_2, \dots, e_n ist, ergibt sich der Abstand zum Kellerpegel zu:

$$q = m + |A| - 1$$

Beachten Sie, dass beim Anlegen eines neuen Objekts bisher nur der dafür benötigte Platz allokiert wurde. Die Initialisierung insbesondere des Verweises auf die Tabelle der überschreibbaren Methoden haben wir an den Konstruktor delegiert.

5.7 Die Definition von Konstruktoren

Als letztes wollen wir noch überlegen, wie wir die Definition eines Konstruktors übersetzen wollen. Im Allgemeinen sieht eine solche Definition so aus:

$$d \equiv A(t_2 x_2, \dots, t_n x_n) \{ ss \}$$

Natürlich wird unsere Vorgehensweise sein, hier nichts Neues zu erfinden, sondern die Definition eines Konstruktors wie die Definition einer normalen Methode zu übersetzen, die keinen Rückgabewert liefert – mit der einzigen Besonderheit, dass sie zuerst das Feld für die Tabelle überschreibbarer Funktionen initialisieren muss. Dazu definieren wir die Makro-Instruktion `initVirtual A`:

```
initVirtual A = loadc  $\_tA$ 
                storem 0
                pop
```

wobei $_tA$ die Anfangsadresse der Tabelle t_A für die überschreibbaren Methoden der Klasse A ist. Dann definieren wir:

```
code  $d \rho =$  enter  $q$ 
              initVirtual  $A$ 
              code  $ss \rho'$ 
```

wobei q der Platzbedarf auf dem Keller und ρ' die Adressumgebung innerhalb des Konstruktors A nach Abarbeitung der formalen Parameter ist.

Beispiel 5.7.1 Bei der Übersetzung der Klasse aus Beispiel 5.2.1 fehlt uns nur noch die Übersetzung des Listen-Konstruktors. Diese liefert den folgenden Code:

```
enter 3      alloc 0      loada 1      loadc 0
loadc  $\_tlist$  loadr  $-4$     loadc 1      storem 2
storem 0     storem 1     add           pop
pop         pop         storea 1     return
                        pop
```

Sie sehen, dass abgesehen von der Initialisierung der Tabelle der überschreibbaren Funktionen nichts Ungewöhnliches passiert ist. Die Referenz auf das Objekt, für den der Konstruktor aufgerufen wird, hat wie bei anderen Methoden die Adresse -3 (relativ zum FP). Entsprechend kann man auf die Attribute dieses Objekts mittels der Instruktionen **loadm** und **storem** zugreifen. \square

Natürlich kann ein Konstruktor Konstruktoren der Oberklasse aufrufen. In C++-Syntax wird dieser im Kopf der Konstruktor-Deklaration angegeben:

$$d \equiv A(t_2\ x_2, \dots, t_m\ x_m) : B(e_2, \dots, e_n) \{ ss \}$$

Eine Übersetzung dieser Deklaration muss dafür sorgen, dass vor der Auswertung des Rumpfs ss der Konstruktor der angegebenen Oberklasse aufgerufen wird. Da dieser für dasselbe Objekt aufgerufen wird wie der aufrufende Konstruktor, wird ihm als Objektreferenz der Verweis auf das aktuelle Objekt übergeben. Auch müssen seine aktuellen Parameter nicht relativ zur Adressumgebung innerhalb des Konstruktors A , sondern relativ zur Adressumgebung der Klasse A , erweitert um die formalen Parameter des Konstruktors ausgewertet werden. Da der Konstruktor der Oberklasse möglicherweise seine eigenen Versionen der überschreibbaren Methoden aufruft, warten wir mit der Initialisierung des Tabellenverweises, bis der Konstruktor der Oberklasse ausgeführt wurde:

```
code  $d\ \rho =$  enter  $q$ 
               codeR  $e_n\ \rho_1$ 
               ...
               codeR  $e_2\ \rho_1$ 
               loadr  $-3$ 
               mark
               loadc  $_B$ 
               call
               initVirtual  $A$ 
               code  $ss\ \rho'$ 
```

wobei q der Platzbedarf des Konstruktors auf dem Keller, ρ_1 die Adressumgebung zur Auswertung der aktuellen Parameter des Konstruktors der Oberklasse und ρ' die Adressumgebung innerhalb des Konstruktors A sind.

Unsere Implementierung des Anlegens neuer Objekte stellt sicher, dass jeder Konstruktor die Versionen virtueller Methoden verwendet, die in seiner Klasse gelten. Um dies zu erreichen, legten wir in jedem neuen Objekt den Verweis auf die aktuelle Tabelle der virtuellen Funktionen erst *nach* dem Aufruf der Konstruktoren der Oberklassen an.

Es sollte jedoch nicht verschwiegen werden, dass das Problem, wann während des Anlegens eines neuen Objekts die Tabelle der virtuellen Methoden anzulegen ist, bei verschiedenen Programmiersprachen unterschiedlich gelöst ist. Im Gegensatz

zu der hier beschriebenen Vorgehensweise kann in Java auch der Konstruktor der Oberklasse B bereits auf die Methoden der Unterklasse A zugreifen. Für Java muss folglich die Tabelle der virtuellen Methoden bereits *vor* dem Konstruktor-Aufruf für die Oberklasse angelegt werden.

5.8 Ausblick: Mehrfache Beerbung

In diesem Abschnitt beschließen wir unsere knappe Einführung in die Übersetzung objektorientierter Programmiersprachen mit einem Ausblick auf Konzepte *mehrfacher Beerbung*. Mehrfache Beerbung wird unterstützt von Programmiersprachen wie C++, Eiffel und Scala. Diese Programmiersprachen erlauben einer Unterklasse A , mehrere Oberklassen B_1, \dots, B_k simultan zu beerben¹ (engl. Multiple Inheritance). Die Herausforderung bei diesem Konzept besteht sowohl in der Erfindung raffinierter Implementierungstechniken, als auch im angemessenen Sprachentwurf selbst: welche Schwierigkeiten auftreten, um die komplexe Semantik der Mehrfachbeerbung in C++ zu formalisieren, zeigt etwa die Diskussion in [WNST06]. Ein fundamentales Problem bereitet die Auflösung der bei der Beerbung möglicherweise auftretenden Mehrdeutigkeiten; so können etwa verschiedene Methoden gleichen Namens geerbt werden oder dieselbe Oberklasse auf mehreren Wegen zu einer Unterklasse A beitragen.

Eine besonders einfache Form von Mehrfachbeerbung findet sich in der Programmiersprache Java. In Java gibt es keine Mehrfachbeerbung im eigentlichen Sinne: neben einer Oberklasse B , von der die Klasse A erbt, kann die Klasse A beliebig viele weitere *Interfaces* implementieren. Ein Interface ist eine *abstrakte* Klasse ohne eigene Attribute, für deren Methoden implementierende Unterklassen Definitionen bereitstellen müssen. Im Java-Jargon heißen solche Methoden *abstrakt*, bei C++ *rein virtuell* (pure virtual).

Da Interfaces über keine Attribute verfügen und in der Klasse A für jede Methode maximal eine Implementierung zur Verfügung steht, gibt es in diesem Fall keine Mehrdeutigkeiten. Es ergibt sich nur eine zusätzliche Schwierigkeit. Nehmen Sie an, ein Methodenaufruf $e.f(\dots)$ soll übersetzt werden, bei dem der statisch bekannte Typ des Ausdrucks e ein Interface I ist. Im Allgemeinen kann nicht garantiert werden, dass die Methode f in sämtlichen das Interface I implementierenden Klassen A' die gleiche Relativadresse hat. Ein Ausweg besteht darum darin, für jede Klasse A eine Hash-Tabelle h_A bereitzustellen, die jedem Namen f einer Methode die jeweils in A gültige Codeadresse der zugehörigen Implementierung zuordnet. Damit diese Hash-Tabelle effizienter implementiert werden kann, stellt Java zusätzlich den *String Pool* zur Verfügung, in den sämtliche statisch bekannten Strings, also insbesondere sämtliche Methodennamen eingetragen werden. Anstelle der textuellen Repräsentation der Namen können dann die Referenzen auf die Repräsentationen als Schlüssel verwendet werden.

Eine alternative Implementierung, mehr gemäß der Philosophie von C++ würde in einem A -Objekt für jedes der implementierten Interfaces I_j einen Verweis auf

¹ Der im Deutschen übliche, jedoch unlogische Begriff lautet hier *Mehrfachvererbung*

eine eigene Methodentabelle t_{A,I_j} für I_j innerhalb der Klasse A unterbringen. In der Tabelle t_{A,I_j} stehen die Verweise auf die Implementierungen der in I_j deklarierten Methoden. Jede Methode f des Interface I_j erhält eine feste Relativadresse, mit der die aktuelle Anfangsadresse einer Implementierung von f in allen Tabellen t_{A',I_j} für implementierende Klassen A' nachgeschlagen werden kann.

Nehmen wir an, ein Ausdruck e habe den statischen Typ $I *$ für ein Interface I und werte sich zur Laufzeit zu einem Zeiger auf ein Objekt o einer Klasse A aus, die I implementiert. Die Methode f eines Aufrufs $e \rightarrow f$ muss dann in der entsprechenden Tabelle $t_{A,I}$ des Objekts o nachgeschlagen werden. Um diese Tabelle schnell zu finden, lassen wir den Zeiger auf das Objekt o nicht auf den Anfang des Speicherbereichs von o zeigen, sondern auf den Verweis auf die Tabelle $t_{A,I}$ (Abb. 5.5). Dieser

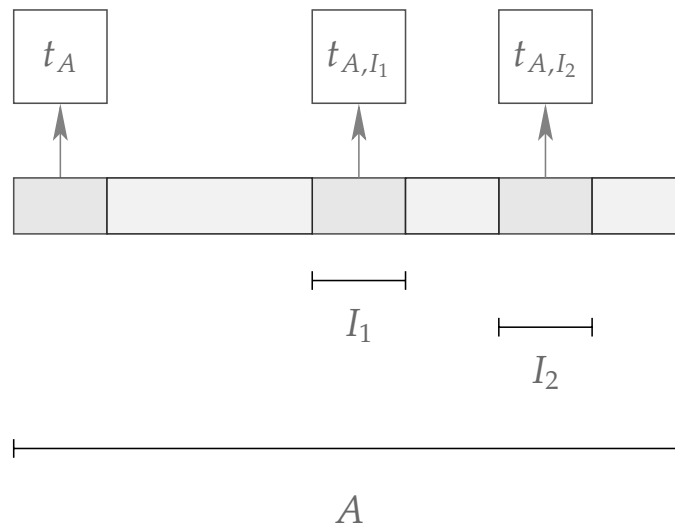


Abb. 5.5. Objekt einer Klasse, das die Interfaces I_1, I_2 implementiert.

Zeiger repräsentiert die *I-Sicht* auf das Objekt o .

Beachten Sie, dass mit dieser Idee zwei Probleme verbunden sind.

- Die in einem Interface deklarierten Methoden werden möglicherweise auf Attribute des Objekts zugreifen wollen. Deshalb müssen wir in der Tabelle für jedes Interface I auch den *Abstand* hinterlegen, den der Anfang des Objekts von dem Verweis auf die Tabelle $t_{A,I}$ besitzt. Dies erlaubt uns, jedem Aufruf einer Methode f den *rekonstruierten* Objektverweis zu übergeben.
- Zum anderen müssen Typanpassungen bei Verweisen auf Objekte die Sichten und damit auch konkret die Verweise ändern. Bei der Zuweisung:

$$I * x \leftarrow \mathbf{new} A();$$

muss vor der Zuweisung die *I-Sicht* auf das neue A -Objekt ausgewählt werden, indem der Zeiger auf das Objekt entsprechend verschoben wird.

Diese zweite Art der Realisierung von Interfaces umgeht die Benutzung von Hash-Tabellen, hat jedoch den Nachteil, dass die Anfangsadresse einer Methode f eventuell mehrmals abgespeichert werden muss: einmal in der Tabelle für A und zusätzlich in der Tabelle für jedes von A implementierte Interface, zu dem f gehört. Eine Verallgemeinerung des eben skizzierten Ansatzes auf allgemeinere Formen von Mehrfachbeerbung diskutieren die Aufgaben 4 und 5.

5.9 Aufgaben

Der folgende Ausschnitt einer C++-Klassenbibliothek graphischer Objekte dient als durchgängiges Beispiel für die Aufgaben dieses Abschnitts.

```
class graphical_object {
    virtual void translate(double x_offset, double y_offset);
    virtual void scale(double factor);
    // evtl.\ weitere allgemeine Methoden der Klasse
};
class point : public graphical_object {
    double      xc, yc;
public:
    void translate(double x_offset, double y_offset) {
        xc+=      x_offset;
        yc+=      y_offset;
    }
    void scale(double factor) {
        xc*=      factor;
        yc*=      factor;
    }
    point(double x0, double y0) { xc= x0; yc= y0; }
    void set(double x0, double y0) { xc= x0; yc= y0; }
    double x(void) { return xc; }
    double y(void) { return yc; }
    double dist(point &);
};
class closed_graphical: public graphical_object {
public:
    // vom Objekt eingeschlossene Flaeche
    virtual double area(void);
};
class ellipse: public closed_graphical {
    point _center;          // Zentrum der Ellipse
    double _x_radius, _y_radius;
                                // Radien der Ellipse
    double _angle;          // Drehung gegen x-Achse
```

```

public:
    // Konstruktor
    ellipse(point &center,
            double x_radius, double y_radius,
            double angle) {
        _center= center;
        _x_radius= x_radius;  _y_radius= y_radius;
        _angel= angle;
    }
    // Ellipsenflaeche -- ueberschreibt 'closed_graphical::area'
    double area(void) { return PI * _x_radius * _y_radius; }
    // Distanz zu einen Punkt -- aufwendig!
    virtual double dist(point &);
    // Zentrum
    point* center(void) { return &_center; }
    // Verschieben -- ueberschreibt 'graphical_object::translate'
    void translate(double x_offset, double y_offset) {
        _center.translate(x_offset, double y_offset);
    }
    // Skalieren -- ueberschreibt 'graphical_object::scale'
    void scale(double scale_factor) {
        _x_radius *= scale_factor;
        _y_radius *= scale_factor;
    }
    // ....
};

class circle: public ellipse {
public:
    // Konstruktor
    circle(point &center, double radius){
        ellipse(center,radius,radius);
    }
    // Distanz zu einem Punkt -- ueberschreibt 'ellipse::dist'
    virtual double dist(point &p) {
        double center_dist= _center.dist(p);
        if (center_dist <= radius) return 0;
        else return center_dist - radius;
    }
    // ....
};

```

1. *Methoden.* Übersetzen Sie die Methode `ellipse :: translate`.
2. *Methoden-Tabellen.* Bestimmen Sie die Methodentabellen und die Methodenindizes für die in der vorherigen Aufgabe definierten Klassen und ihre Methoden

und übersetzen Sie einen Methodenaufruf der Form $c.dist(p)$, wobei c ein Kreis und p ein Punkt ist.

3. *Teiltypen.* Die Neudefinition einer Klasse soll keine Auswirkungen auf das bestehende Klassengefüge verursachen. Aufgrund der Teiltypregel müssen jedoch auch Methoden schon bestehender Klassen mit Objekten der neuen Klasse arbeiten können. Durch Überschreiben von Merkmalen beerbter Klassen kann deren Sicht auf die neuen Objekte inkonsistent werden. Redefinitionen sind deshalb nur eingeschränkt erlaubt. Drei unterschiedliche Aspekte müssen beachtet werden.

- *Die Bedeutung (Semantik) der Merkmale:*
jedes Attribut und jede Methode hat eine Aufgabe, die durch das Überschreiben nicht beeinträchtigt werden darf. So hat etwa die Methode *scale* die Aufgabe, ein graphisches Objekt zu skalieren. Eine Redefinition muss dieselbe Aufgabe erfüllen, sowohl auf der aktuellen Ebene wie auf der Ebene einer Oberklasse betrachtet.
- *Die Einschränkungen durch das Typsystem:*
Redefinitionen dürfen nicht zu Typinkonsistenzen führen.
- *Die Einschränkungen des Übersetzungsschemas:*
Redefinitionen können die bei der Übersetzung gemachten Annahmen ungültig werden lassen und dürfen dann nicht zugelassen werden.

Der erste Aspekt ist zwar wesentlich, jedoch verfügt ein Übersetzer normalerweise nicht über die notwendige Spezifikation, um eine semantische Zulässigkeit zu verifizieren. In dieser Aufgabe betrachten wir Einschränkungen durch das Typsystem.

Der Übergang von einem Typ zu einem Teiltyp heißt *Typverschärfung*, der Übergang von einem Typ zu einem Obertyp *Typabschwächung*. Ein Prototyp (einer Methode) wird verschärft, wenn die Typen des Rückgabewertes und der Ausgabeparameter verschärft und die Typen der Eingabeparameter abgeschwächt werden.

- a) Begründen Sie, dass aus Sicht des Typsystems keine Einwände gegen eine Verschärfung des Prototyps einer redefinierten Methode bestehen.
Zeigen Sie an Hand von Beispielen, dass der Prototyp öffentlicher, also von beliebigen Klassen aktivierbarer Methoden höchstens verschärft werden kann; jede andere Abänderung kann zu Typfehlern bei bisher korrekten Aufrufen führen.
Begründen Sie, dass eine nicht verschärfende Prototypänderung auch für redefinierte private Methoden nur unter sehr eingeschränkten Bedingungen zulässig ist. Zur Verifikation der Zulässigkeit ist Information über die Oberklassen notwendig, die über die Kenntnis der Typen ihrer Attribute und Prototypen ihrer Methoden hinausgeht – welche?
- b) Die Sprache Eiffel erlaubt in einer abgeleiteten Klasse, den Typ eines geerbten Attributes zu verschärfen. Begründen Sie, warum Attribute dann von einer fremden Klasse nur gelesen werden dürfen.
Zeigen Sie anhand eines Beispiels, dass die Verschärfung des Typs eines Attributes nur unter sehr eingeschränkten Bedingungen zulässig ist. Zur Ve-

rifikation der Zulässigkeit ist Information über die Oberklassen notwendig, die über die Kenntnis der Typen ihrer Attribute und Prototypen ihrer Methoden hinausgeht – welche?

4. *Mehrfachbeerbung I.* In dieser Aufgabe behandeln wir Mehrfachbeerbung. Zur Vereinfachung nehmen wir an, dass von jeder Oberklasse B genau eine Instanz an eine Unterklasse A vererbt wird, jedes A -Objekt also genau ein B -Teilobjekt enthält.

Für jede Oberklasse B von A sehen wir einen festen Speicherbereich innerhalb des Speicherbereichs eines Objekts der Klasse A vor. Dabei nehmen wir an, dass die jeweils erste unmittelbare Oberklasse B_1 so behandelt wird wie die unmittelbare Oberklasse bei einfacher Beerbung: ihr Speicherbereich soll am Anfang des Speicherbereichs für die unmittelbare Unterklasse A liegen (Abb. 5.6).

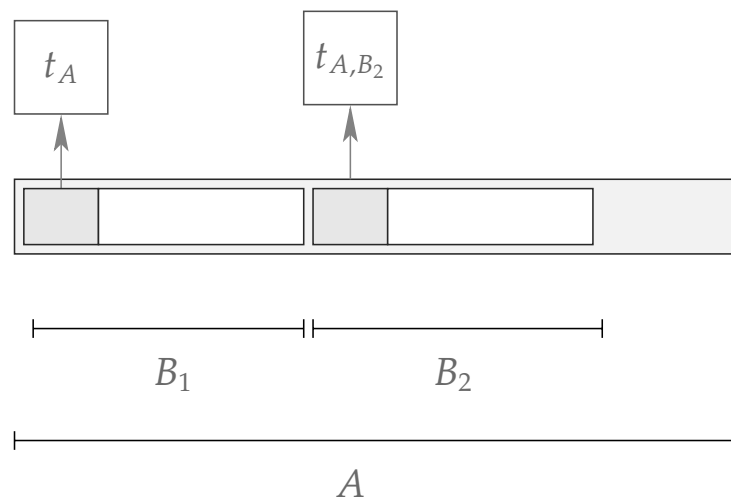


Abb. 5.6. Die Objektorganisation für zwei Oberklassen.

Für eine Oberklasse B erhalten wir die B -Sicht auf ein Objekt der Klasse A , indem wir einen Zeiger auf den Anfang des für B vorgesehenen Bereichs setzen. Insbesondere stimmt damit die A -Sicht auf das Objekt mit der B_1 -Sicht für die erste unmittelbare Oberklasse B_1 überein.

Innerhalb des Speicherbereichs für A legen wir wie bei Einfachbeerbung als erstes die Tabelle t_A aller in A bekannten virtuellen Funktionen ab. Genauso legen wir am Anfang aller nicht-ersten unmittelbaren Oberklassen B einen Verweis auf eine Tabelle $t_{A,B}$ ab, die die Anfangsadressen aller in B bekannten Methoden von A enthält. Dies setzen wir entsprechend für alle Oberklassen der unmittelbaren Oberklassen B von A fort.

Eine in B bekannte Methode f kann jedoch von einer Unterklasse A' von B überschrieben worden sein (welche A selbst oder eine Oberklasse von A ist). Zusätzlich zu der Anfangsadresse von f müssen wir deshalb in $t_{A,B}$ auch den Abstand zwischen den Speicherbereichen für die Klassen A' und B vermerken.

Dies erlaubt uns, vor dem Aufruf von f von B aus die richtige A' -Sicht auf das A -Objekt zu konstruieren.

- a) Führen Sie die Details der Definitionen der Tabellen $t_{A,B}$ aus! Implementieren Sie die Typanpassung eines Zeigers auf ein A -Objekt an einen Zeiger auf ein Objekt einer Ober- bzw. Unterklasse.
 - b) Diskutieren Sie Strategien zur Lösung von Namenskonflikten und die dafür notwendigen Implementierungen!
 - c) Kombinieren Sie den Grundansatz zur Mehrfachbeerbung mit unseren früheren Übersetzungsschemata.
 - d) Erproben Sie Ihre neuen Übersetzungsschemata an unserer Beispiel-Klassenbibliothek!
5. *Mehrfachbeerbung II.* In dieser Aufgabe betrachten wir erneut Mehrfachbeerbung. Wie in Aufgabe 4 nehmen wir zur Vereinfachung an, dass von jeder Oberklasse B genau eine Instanz an die Unterklasse A vererbt wird. Die Idee der Objektorganisation in Aufg. 4 war, dass für jede nicht-erste Oberklasse B eine eigene Tabelle $t_{A,B}$ die in B sichtbaren virtuellen Methoden der Klasse A sammelt. Sämtliche Methoden aus $t_{A,B}$ sind damit insbesondere auch in t_A vermerkt. Diese Ineffizienz möchten wir beseitigen.
- Unser Ziel ist, eine in B bekannte virtuelle Methode f nur in der Tabelle $t_{A,B}$ abzuspeichern, nicht dagegen in den Tabellen t_A und $t_{A',B}$ für Klassen A' zwischen A und B . Dazu modifizieren wir die Adressumgebung für Klassen A so, dass $\rho_A(f)$ nun die Relativadresse d_t des Verweises auf die Tabelle t innerhalb des Speicherbereichs für A liefert zusammen mit dem Index i , an dem der Eintrag für f in t vermerkt ist. Relativ zu einer A -Sicht a auf ein Objekt erhalten wir dann Anfangsadresse des Paares $(_f, Abstand)$ für die Methode f auf die folgende Weise:
- $$S[a + d_t] + 2 \cdot i$$
- (a) Modifizieren Sie den Aufruf einer virtuellen Methode entsprechend dieser Optimierung.
 - (b) Schätzen Sie die Einsparung ab, die diese Optimierung bietet.
 - (c) Geben Sie einen Algorithmus an, der zu einer gegebenen Klassenhierarchie die Abstände $d_{A,B}$, die Adressumgebungen ρ_A für virtuelle Funktionen sowie die benötigten Tabellen berechnet.
6. *Generics.* Implementieren Sie eine generische Klasse für Warteschlangen, d.h. first-in-first-out Queues. Erläutern Sie, wie ein Übersetzer mit mehreren Instanzen Ihrer Warteschlange, etwa `queue<int>` und `queue<list>` umgehen soll!

5.10 Liste zusätzlicher Register

COP, Zeiger auf aktuelles Objekt (Current Object Pointer) S. 167

5.11 CMa-Instruktionen für Objekte

loadm	S. 168	loadsc	S. 169
loadmc	S. 168	loadv	S. 169
loads	S. 170	storem	S. 168

5.12 Literaturhinweise

Simula67, der Urvater objektorientierter Sprachen, ist in [DN66] und in [Sim87] beschrieben. Die Standardreferenz für Smalltalk ist [GR83]; Smalltalk und die zugehörige virtuelle Maschine VisualWorks findet man in [GHH98]. C++ wird in [Str00] definiert. Die Grundlage für die Standardisierung von C++ durch ANSI ist [ES90]. [Mey88] und [Mey92] geben eine gute Einführung in Eiffel. [Cox86] beschreibt Objective-C. Objektorientierte Spracherweiterungen von Lisp sind in [BS82, Can80] beschrieben. Eine interessante moderne objektorientierte Programmiersprache ist Scala [OAC⁺04].

Über Java und C# gibt es mittlerweile zahllose Bücher. Einen Überblick über die Implementierung objektorientierter Programmiersprachen wie Smalltalk, Java oder C++ gibt [BH98]. Die technische Grundlage der gängigen virtuellen Maschine für Java erläutert die Java Virtual Machine Spezifikation [LY99]. Krall präsentiert Techniken zur Just-In-Time Compilierung [Kra98]. Stärk et al. [SSB01] bieten eine Formalisierung der JVM und diskutieren Korrektheitsfragen.

Einen weiteren Ansatz zur Implementierung von Java oder C# bietet das .NET Framework mit der Common Language Runtime (CLR) [ECM06, MG00]. Einen Vergleich dieser beiden Ansätze versuchen die Arbeiten [Gou01, Sin03].

Literaturverzeichnis

- [ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. *Web Services*. Springer Verlag, 2003.
- [AK91] H. Ait-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. MIT Press, 1991.
- [Amm81] U. Ammann. *Code Generation of a Pascal-Compiler*. In [Bar81], 1981.
- [Apt90] K.R. Apt. *Logic Programming, Handbook of Theoretical Computer Science*. Elsevier, 1990.
- [AW06] Krzysztof Apt, Mark Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2006.
- [Bar81] D.W. Barron (Hrsg.). *Pascal — The Language and its Implementation*. Wiley, 1981.
- [Bez88] M. Bezem. *Logic Programming and PROLOG*. In CWI Quarterly **1**(3), pp. 15–29, Amsterdam, Centre for Mathematics and Computer Science, 1988.
- [BH98] Bernhard Bauer, Riitta Höllerer. *Übersetzung objektorientierter Programmiersprachen: Konzepte, abstrakte Maschinen und Praktikum*. Springer Verlag, 1998.
- [Bru04] Michael Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.
- [BS82] Daniel G. Bobrow, Mark J. Stefik. *LOOPS: an Object-Oriented Programming System for Interlisp*, 1982.
- [Can80] H. I. Cannon. *Flavors*. Technical report, MIT Artificial Intelligence Laboratory, 1980.
- [CM03] W.F. Clocksin, C.S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, 2003.
- [Cox86] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.
- [DN66] Ole-Johan Dahl, Kristen Nygaard. *Simula - An Algol-based Simulation Language*. *Communications of the ACM (CACM)*, 9(9):671–678, 1966.
- [ECM06] Common Language Infrastructure (CLI). 4th Edition. Technical Report ECMA-335, ECMA International, 2006.
- [ES90] Margaret A. Ellis, Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [FW87] J. Fairbairn, S.C. Wray. *TIM. A Simple, Lazy Abstract Machine to Execute Supercombinators*. In Proc. Functional Programming Languages and Computer Architecture, LNCS 274, pp. 34–45. Springer, 1987.

- [GHH98] Adele Goldberg, Timothy Howard, Tim Howard. *The Smalltalk Developer's Guide to VisualWorks with Disk*. Advances in Object Technology. Cambridge University Press, 1998.
- [Gou01] K. John Gough. *Stacking them up: a Comparison of Virtual Machines*. In 6th Australasian Computer Systems Architecture Conference (ACSAC), pp. 55–61, 2001.
- [GR83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [Han86] M. Hanus. *Problemlösen mit Prolog*. Teubner Verlag, 1986.
- [Hau06] Tobias Hauser. *Einstieg in ActionScript*. Galileo Press, 2006.
- [HS99] Michael Hanus, Ramin Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Special Issue 1, 1999.
- [Inc99] Adobe Systems Inc. *PostScript(R) Language Reference (3rd Edition)*. Addison-Wesley, 1999.
- [JL96] Richard Jones, Rafael Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996.
- [Joh84] T. Johnsson. *Efficient Compilation of Lazy Evaluation*. In Proc. ACM SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices **19**(6), pp. 58–69, 1984.
- [Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming (JFP)*, 2(2):127–202, 1992.
- [Kay04] Michael Kay. *XSLT 2.0 Programmer's Reference (3rd Edition)*. Wrox, 2004.
- [KBS86] H. Kleine-Büning, S. Schmittgen. *Prolog*. Teubner, 1986.
- [KCD⁺03] Howard Katz, Don Chamberlin, Denise Draper, Mary Fernandez, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, Philip Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.
- [Kra98] Andreas Krall. *Efficient JavaVM Just-in-Time Compilation*. In IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 205–212, 1998.
- [Lan64] P.J. Landin. *The Mechanical Evaluation of Expressions*. In Computer Journal **6**(4), 1964.
- [Ler90] Xavier Leroy. The ZINC Experiment : An Economical Implementation of the ML Language. Technical Report RT-0117, INRIA, Rocquencourt, February 1990.
- [Llo87] J.W. Lloyd. *Foundations of Logic Programming*. 2. Auflage, Springer, 1987.
- [LY99] Tim Lindholm, Frank Yellin. *Java Virtual Maschine Specification. Second Edition*. SUN Microsystems Inc., 1999.
- [Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
- [Mey92] Bertrand Meyer. *Eiffel. The Language*. Prentice-Hall, 1992.
- [MG00] Eric Meijer, Jeremy Gough. Technical Overview of the Common Language Runtime, 2000.
- [MW88] D. Maier, D.S. Warren. *Computing with Logic, Logic Programming with Prolog*. Benjamin/Cummings, 1988.
- [OAC⁺04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger. An Overview of the Scala Programming Language. Technical report, EPFL-LAMP, 2004.

- [PD82] St. Pemberton, M. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.
- [Ped04] Volnei A. Pedroni. *Circuit Design with VHDL*. MIT Press, 2004.
- [PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- [RR64] B. Randell, L.J. Russel. *Algol60 Implementation*. Academic Press, 1964.
- [Sco05] Michael L. Scott. *Programming Language Pragmatics (2nd Edition)*. Morgan Kaufmann, 2005.
- [Seb05] Robert W. Sebesta. *Concepts of Programming Languages (7th Edition)*. Addison-Wesley, 2005.
- [SIC06] SICStus 4 beta documentation, 2006. Homepage: <http://www.sics.se/isl/sicstuswww/site/>.
- [Sim87] SIS, *Data Processing Programming Languages — SIMULA*, 1987. Svensk Standard SS 636114.
- [Sin03] Jeremy Singer. JVM versus CLR: A Comparative Study. *Proceedings of the 2nd Int. Conf. on Principles and Practice of Programming in Java*, pp. 167–169, 2003.
- [SN05] Jim Smith, Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2005.
- [SS94] L. Sterling, E. Shapiro. *The Art of Prolog, Advanced Programming Techniques*. MIT Press, 1994.
- [SSB01] Roland Stärk, Joachim Schmid, Egon Börger. *Java and the Virtual Machine Definition, Verification, Validation*. Springer Verlag, 2001.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language. Special Edition*. Addison-Wesley, 2000.
- [SWD05] Tom Schrijvers, Jan Wielemaker, Bart Demoen. *Constraint Handling Rules for SWI-Prolog*. In Workshop on (Constraint) Logic Programming, Ulm, 2005.
- [TN06] Allen B Tucker, Robert Noonan. *Programming Languages*. McGraw-Hill-Science, 2006.
- [VSD02] Ruben Vandeginste, Konstantinos F. Sagonas, Bart Demoen. *Segment Order Preserving and Generational Garbage Collection for Prolog*. In Practical Aspects of Declarative Languages, 4th Int. Symposium (PADL), pp. 299–317. Springer, LNCS 2257, 2002.
- [War77] D.H.D. Warren. *Applied Logic — Its Use and Implementation as a Programming Language Tool*. Ph.D. Thesis, Univ. of Edinburgh, 1977.
- [War83] D.H.D. Warren. *An Abstract PROLOG Instruction Set*. Technical Note 309, SRI International, 1983.
- [Wil92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. In Proc. Int. Workshop on Memory Management (IWMM), pp. 1–42. Springer, LNCS 637, 1992.
- [WNST06] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, Frank Tip. *An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++*. In 21st Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 345–362, 2006.

Sachverzeichnis

- Abschluss, 65, 89
 - Code Pointer, 67
 - Global Pointer, 67
- Abstraktion
 - Daten-, 159
 - funktionale, 8, 59
- ActionScript, 2
- Ada, 1, 163
- Adressoperator &, 30
- Adressumgebung, 12
 - CMa, 23
 - MaMa, 68
 - WiM, 114
- Algol, 1, 35, 63, 164
- Algol Object Code, 58
- Anfrage, 112, 137
- Anweisung, 7, 15
 - do-while*, 16
 - for*, 16
 - goto*, 8
 - if*, 16
 - switch*, 20
 - while*, 16
- Attribut, 160
- Aufrufbaum, 34
- Ausdruck
 - FuL, 62
 - C, 8, 10
 - Linkswert, 13
 - Rechtswert, 13
- Auswertung
 - Applicative Order, 63
 - Normal Order, 64
 - partielle, 3
 - verzögerte, 64
- Backtrack Point, 132
- Backtrack Pointer *BP*
 - WiM, 132
- Backtracking, 112, 132
- Basisklasse, 161
- bedingte Anweisung, 16
- Beerbung, 161
- Befehlszähler *PC*
 - CMa, 9
 - MaMa, 66
 - WiM, 112
- Berechnung
 - gemischte, 3
- Bindung
 - statische/dynamische
 - CMa, 36
 - FuL, 63, 64
 - OMa, 162
- C, 1, 7, 35
- C++, 164
- CMa, 7
 - check, 21
 - Adressumgebung, 23
 - Befehlszähler *PC*, 9
 - Datenspeicher, 8
 - Extreme Stack Pointer *EP*, 29
 - Frame Pointer *FP*, 37
 - Halde, 28
 - Haldenzeiger *HP*, 28
 - Hauptzyklus, 9

- Heap Pointer *HP*, 28
- Keller, 8, 37
- Kellerrahmen, 37
- Kellerzeiger *SP*, 8
- lokaler Keller, 38
- organisatorische Zellen, 38
- Programmspeicher *C*, 9
- Rahmenzeiger *FP*, 37
- Relativadresse, 12, 23, 25
- Rücksprungadresse, 37
- Schrankenzeiger *EP*, 29
- Stack Pointer *SP*, 8
- Variable, 12
- Cobol, 1
- Codeerzeugung *CMA*
 - code, 13
 - code_L, 13
 - code_R, 13
- Codeerzeugung *MaMa*
 - code_B, 69
 - code_C, 68
 - code_V, 67
- Codeerzeugung *WiM*
 - code_A, 114
 - code_C, 130
 - code_G, 118
 - code_P, 131
 - code_U, 125
- Current Object Pointer, 167
- Cut-Operator, 147
- Dangling Reference, 33
- Datenbereich, 38
- Datenspeicher
 - CMA*, 8
- Dereferenzierung, 30
- Dynamic Scoping, 64
- Eiffel, 164
- elab_formals, 40
- elab_globals, 40
- elab_locals, 40
- Extreme Stack Pointer *EP*
 - CMA*, 29
- Fallunterscheidung
 - MaMa*, 97
- Feld
 - statisches, 23
- Flavors, 164
- Folgerung, 109
- Fortran, 1
- Fortsetzungsadresse
 - negative, 133
 - positive, 119
- Frame Pointer *FP*
 - CMA*, 37
 - MaMa*, 66
 - WiM*, 113
- Freigabe von Speicher, 33
- FuL, 62
- Funktion
 - C*, 34
 - Deklaration, 34
 - Rumpf einer, 34
 - funktionale
 - anonyme, 59
 - Argument Pointer, 67
 - Code Pointer, 67
 - Global Pointer, 67
 - höhere, 60
 - Übersversorgung, 84
 - Unterversorgung, 83
 - funktionale Sprache, 1
- Funktionsaufruf, 34, 42, 79
 - letzter, 101
- Funktionsdefinition, 59, 76
- Garbage Collection, 28, 149
- Generizität, 162
- Global Pointer *GP*
 - MaMa*, 71
- Global-Vektor
 - MaMa*, 71
- Global-Zeiger *GP*
 - MaMa*, 71
- goto*-Anweisung, 8
- Gültigkeitsbereich
 - C*, 36
- Halde
 - CMA*, 28
 - MaMa*, 66
 - WiM*, 113
- Haldenzeiger *HP*
 - CMA*, 28
 - WiM*, 113
- Haskell, 1, 59

- Hauptzyklus
 - CMa, 9
 - MaMa, 66
 - WiM, 112
- Heap Pointer *HP*
 - CMa, 28
 - WiM, 113
- Horn-Klausel, 109
- imperative Sprache, 1
- Implementierung, 2
- Implikation, 1
- indizierter Sprung, 20
- Information
 - dynamische, 22
 - statische, 22
- Informationskapselung, 163
- Inkarnation, 7, 34, 35
 - lebendige, 34
- Inkarnationsweg, 34
- Interface, 175
- Interpreter, 2
- iterative Anweisung, 16
- Java Virtual Machine, 58
- JIT-Übersetzer, 5
- Keller
 - CMa, 8, 37
 - MaMa, 66
 - WiM, 113
- Kellerpegel
 - MaMa, 69, 72
- Kellerrahmen
 - CMa, 37
 - MaMa, 71
 - WiM, 119
- Kellerüberlauf, 29
- Kellerzeiger *SP*
 - CMa, 8
 - MaMa, 66
 - WiM, 113
- Klasse, 160, 161
 - abgeleitete, 161
 - abstrakte, 161, 163, 175
 - Basis-, 161
 - generische, 163
 - parametrisierte, 163
- Klausel, 112, 130
- Horn-, 109
- Indizierung, 144
- Konstantentabelle, 50
- Kontrollfluss, 8
- Laufzeit, 3, 22
- let*-Ausdruck, 75
- letrec*-Ausdruck, 86
- letztes Ziel, 140
- Lisp, 1, 36, 59
- Listen, 97
- Literal, 112, 118
- logische Sprache, 1
- lokaler Keller
 - CMa, 38
 - MaMa, 69
 - WiM, 113
- Loops, 164
- MaMa, 66
 - Adressumgebung, 68
 - Befehlszähler *PC*, 66
 - Frame Pointer *FP*, 66
 - getvar*, 74
 - Global Pointer *GP*, 71
 - Global-Zeiger *GP*, 71
 - Halde, 66
 - Hauptzyklus, 66
 - Keller, 66
 - Kellerrahmen, 71
 - Kellerzeiger *SP*, 66
 - organisatorische Zellen, 80
 - Programmspeicher *C*, 66
 - Rahmenzeiger *FP*, 66
 - Relativadresse, 72
 - Rücksprungadresse, 81
 - Stack Pointer *SP*, 66
- Maschine
 - real, 4
 - reale, 7
 - virtuelle, 4
 - CMa, 7
 - MaMa, 66
 - WiM, 112
- match*-Ausdruck, 97
- Mehrfachbearbung, 175
- Merkmal, 160
 - öffentliches, 163
 - privates, 163

- Methode, 160
 - abstrakte, 175
 - Aufruf, 168
 - rein virtuelle, 175
 - virtuelle, 165
- Modula, 1
- Modularisierung, 159
- Name, 7
 - angewandtes Vorkommen, 35
 - definierendes Vorkommen, 35
 - freier, 36
 - gebundener, 35
 - globaler, 35
 - lokaler, 35
- Negation
 - Prolog, 148
- negative Fortsetzungsadresse, 133
- Negierung durch Fehlschlag, 148
- Objekt, 159
 - klasse, 160
 - anonymes, 28
 - Attribut, 160
 - eigentliches, 162
 - Klasse, 163
 - Merkmal, 160
 - Methode, 160
- Objektklasse, 161
- objektorientierte Sprache, 2
- OCaml, 1, 59
- Occur-Check, 123
- organisatorische Zellen
 - CMa, 38
 - MaMa, 80
 - WiM, 119
- Parameter
 - aktueller, 43
 - formaler, 34
 - Übergabe
 - FuL, 63
 - Übergabe
 - C, 43
- Parameterübergabe
 - Call-by-Name, 64
 - Call-by-Need, 64
 - Call-by-Reference, 45
 - Call-by-Value, 45, 63
- Pascal, 1, 63
- Pattern Matching, 60, 96
- PL/I, 35
- Pointer, 28, 29
- Polymorphie, 61, 162
- positive Fortsetzungsadresse, 119
- Postscript, 2
- Prädikat, 131
- Prädikatenlogik, 1, 109
- Program Counter *PC*
 - WiM, 112
- Programmausdruck, 94
- Programmeinheit, 36
- Programmentwicklung, 160
- Programmiersprache
 - funktionale, 1, 59
 - für spezielle Anwendungen, 2
 - höhere, 1
 - imperative, 1, 7
 - logische, 1, 109
 - objektorientierte, 2, 159
 - problemorientierte, 1
- Programmspeicher *C*
 - CMa, 9
 - MaMa, 66
 - WiM, 112
- ProL, 109
- Prolog, 109
- Prozedur, 34
- Quellprogramm, 3
- Quellsprache, 3
- Rahmenzeiger *FP*
 - CMa, 37
 - MaMa, 66
 - WiM, 113
- Reduktion, 1
- referentielle Transparenz, 65
- Register
 - BP*, 132
 - EP*, 29
 - FP*, 66, 113
 - GP*, 71
 - HP*, 28, 113
 - PC*, 9, 66, 112
 - SP*, 8, 66, 113
 - TP*, 132
 - COP*, 167

- Rekursionsgleichung, 60
- rekursive Datentypen, 97
- Relativadresse
 - CMa, 12, 23, 25
 - MaMa, 72
 - WiM, 119
- Resolution, 1, 109
- Rücksetzen, 112, 132
- Rücksetzpunkt, 132
- Rücksetzzeiger *BP*
 - WiM, 132
- Rücksprungadresse
 - CMa, 37
 - MaMa, 81
 - WiM, 119
- Sandkasten-Prinzip, 5
- Schrankenzeiger *EP*
 - CMa, 29
- Seiteneffekt, 112
- Selbstreferenz, 114
- Semantik
 - operationale, 109
- Sichtbarkeit, 36
- Simula 67, 164
- sizeof, 24
- Smalltalk-80, 164
- SML, 1
- Space Leaks, 92
- Speicherallokation, 30
- Speicherbelegungsfunktion, 12
- Speicherbereinigung, 28, 149
- Sprung
 - bedingter, 16
 - indizierter, 20, 145
 - unbedingter, 16
- Spurzeiger *TP*
 - WiM, 132
- Stack Overflow, 29
- Stack Pointer *SP*
 - CMa, 8
 - MaMa, 66
 - WiM, 113
- Static Scoping, 64
- String Pool, 175
- Substitution, 121
 - allgemeinste unifizierende, 121
- Teiltyp, 161
- Teiltypregel, 161
- Term, 112
- Trail
 - WiM, 132
- Trail Pointer *TP*
 - WiM, 132
- try-Kette, 145
- Tupel, 95
- Typ
 - abschwächung, 179
 - anpassung, 162
 - inferenz, 61
 - schema, 61
 - variable, 61
 - verschärfung, 179
 - eigentlicher, 162
 - einfacher, 23
 - Feld, 23
 - zusammengesetzter, 23
- Übersetzer, 3
 - JIT-, 5
- Übersetzungszeit, 3, 22
- Umgebung, 65
- Unifier, 121
 - Most General, 121
- Unifikation, 112, 120
- Unterklasse, 161
- Variable, 7, 59, 112
 - anonyme, 115
 - CMa, 12
 - freie, 63
 - globale, 36
 - initialisierte, 115
 - Lebensdauer, 35, 143
 - lokale, 130
 - MaMa, 71
 - ungebundene, 114
 - uninitialisierte, 115
 - WiM, 114
- Variablendeklaration
 - globale, 40
 - lokale, 40
- Vererbung, 161
- Vererbungshierarchie, 163
- von Neumann-Rechner, 1, 159
- Vorberechnung, 3
- Vorkommenstest, 123

Wert, 63, 112
 Wertzuweisung, 7, 14
 Wiederverwendbarkeit, 159

WiM

try-Kette, 135
 Adressumgebung, 114
 Aufbaublock, 127
 Backtrack Pointer *BP*, 132
 Befehlszähler *PC*, 112
 Frame Pointer *FP*, 113
 Halde, 113
 Haldenzeiger *HP*, 113
 Hauptzyklus, 112
 Heap Pointer *HP*, 113
 Keller, 113
 Kellerrahmen, 119
 Kellerzeiger *SP*, 113
 organisatorische Zellen, 119
 Programmspeicher *C*, 112

Rahmenzeiger *FP*, 113
 Relativadresse, 119
 Rücksetzzeiger *BP*, 132
 Rücksprungadresse, 119
 Spurzeiger *TP*, 132
 Stack Pointer *SP*, 113
 Trail, 132
 Trail Pointer *TP*, 132
 Variable, 114

XQuery, 2
 XSLT-2.0, 2

Zeiger, 28, 29
 -arithmetik, 30
 Ziel, 112
 letztes, 140
 Zielprogramm, 3
 Zielsprache, 3