Von Regeln zu Klassen
Objektorientierte Implementation eines Scheme–Interpreters in Java

Marvin Sielenkemper

Diplomarbeit

vorgelegt

15. Januar 2003

am

Institut für Informatik

der

Westfälischen Wilhelms Universität
Münster

Inhaltsverzeichnis

Abbildungsverzeichnis

0 Einführung

Diese Arbeit beschreibt den Aufbau und die Implementation eines Interpreters für Scheme in Java. Ziele sind

- einen den Anforderungen des *Revised^5 Report on the Algorithmic Language Scheme* [19] genügenden Interpreter zu entwickeln und zu diesem Zweck
- ein klares, modulares und objektorientiertes Design zu entwerfen und in Java zu implementieren.

Den Anstoß gab die Interpreterbau–Vorlesung von Professor Clausing an der Westfälischen–Wilhelms–Universität in Münster im Sommersemester 1998, in der ebenfalls ein Scheme–Interpreter in Java implementiert wurde.

Der dort entwickelte Interpreter bildete den Kontrollfluß eines Scheme–Programms auf den des Java–Programms ab. Damit stehen natürlich nur die Kontrollflußoperationen zur Verfügung, welche die Java–Maschine bietet. Diese Maschine führt mit einem Stack von Aktivierungsblocks Buch über aufgerufene Funktion. Sie kann

- eine Folge von Anweisungen hintereinander ausführen,
- in Abhängigkeit von einer Laufzeit–Berechnung verzweigen (auch zurück, dh es gibt Iterationskonstrukte),
- eine Funktion aufrufen, was einen neuen Aktivierungsblock auf dem Stack ablegt,
- mit `return` eine zuvor aufgerufenen Funktion verlassen, damit wird der oberste Aktivierungsblock vom Stack entfernt und von seinem Nachfolger in dieser Rolle abgelöst,
- mit `catch` einen Aktivierungsblock markieren und
- mit `throw` einen passend markierten Aktivierungsblock im Stack suchen und wieder zum obersten zu machen, dazu werden ggf mehrere darüber liegende entfernt.

Sie kann aber nicht

- den Zustand des Stacks abspeichern und ggf mehrfach wieder herstellen.

Solche Manipulationen sind in der Java–Maschine nicht vorgesehen.

Aus diesem Grunde war es nicht möglich, Fortsetzungen (engl. Continuations) als Werte zu implementieren, die allen Anforderungen des Scheme–Reports genügen. Solche Fortsetzungen sind aber ein wesentlicher Baustein für die Flexibilität und Mächtigkeit von Scheme.

Eine Lösung für dieses Problem fand sich in dem Artikel *Proper Tail Recursion and Space Efficiency* [8] von William Clinger. Er definiert darin Syntax und Semantik eines Kern–Schemes, um dieses dann auf seine Speichereffizienz zu untersuchen. Die Definition ist in Form von Übergangsregeln für Konfigurationen einer abstrakten Maschine angegeben. Diese Maschine wird hier im Sinne einer Implementation erweitert und anschließend implementiert.

Die Arbeit ist wie folgt aufgebaut. In Kapitel 1 wird ein grober Überblick über die Implementation gegeben. In Kapitel 2 werden zunächst die Grundstrukturen eingeführt, aus denen sich die Implementation zusammensetzt. Zuerst wird der Aufbau der Werte, dann der Programme und abschließend der Maschine vorgestellt. Anschließend beschreibt es mit Hilfe einer formalen Semantik die Interpretation von Programmen der Zwischensprache durch Manipulation des Zustands der Maschine. In Kapitel 3 wird ein Überblick über die Implementation gegeben. Der vollständige Quellcode liegt auf der beiliegenden CD vor. Zum Abschluß gibt Kapitel 4 einen Ausblick auf mögliche Ergänzungen.

1 Überblick

In jedem Datenverarbeitungs–System gibt es Daten, die verarbeitet werden, Programme, die die Verarbeitung steuern und einen Interpreter, der die Verarbeitung schließlich durchführt – das kann auch eine CPU sein. Das gilt natürlich auch für ein Scheme–System. Allerdings ist die Syntax der Sprache Scheme so konstruiert, dass jedes Programm auch als Datum gelesen werden kann. Darüber hinaus erlaubt es das laufende Scheme–System, Daten in Programme zu übersetzen und auszuführen. Dadurch verschwimmt die in anderen Sprachen so klare Trennlinie zwischen Daten und Programmen in Scheme ein wenig.

Das erlaubt interessante Anwendungen. Programme können sich selbst ergänzen, Funktionen können als "Datenlückentext" vorbereitet werden, der zur Benutzung geeignet gefüllt wird, usw.

Diese Flexibilität und Vereinheitlichung verursacht allerdings einige Probleme bei der Implementation eines Scheme–Systems. Denn um Programme effizient ausführen zu können, müssen sie intern eben doch anders als Daten behandelt werden: nicht jedes Datum ein Programm. Diese Tatsache sollte beim Entwurf der internen Darstellung beachtet werden.

1.1 Interpretation von Daten als Programm

Es ist möglich, Daten direkt als Programme zu interpretieren – so geschehen im Interpreter der Eingangs erwähnten Vorlesung – denn die Menge der Daten enthält schließlich auch alle korrekten Programme. Allerdings enthält sie zusätzlich noch sehr viele fehlerhaft Programme. Solche syntaktisch falschen Eingaben werden bei der direkten Interpretation erst im Laufe einer Auswertung entdeckt. Der Interpreter muß sie erkennen und darauf reagieren. Dadurch wird er komplexer und langsamer.

Um diese Nachteile zu vermeiden, wurde dem vorliegenden System eine Sprache für Programme hinzugefügt. Dadurch konnte der Daten–Interpreter in zwei Teile zerlegt werden. Der erste ist ein Syntax–Prüfer und Übersetzer. Er tritt für jedes Programm nur einmal in Aktion. Der zweite Teil ist ein Interpreter. Er führt die im geprüften Programm vorgegebenen Aktionen aus und kann dabei sicher sein, nicht auf Syntax–Fehler zu stoßen. Die Interpretation ist so einfacher und geschieht schneller.

1.2 Aufbau

Hier werden kurz die verschiedenen Funktionseinheiten und die Sprachen in denen sie miteinander kommunizieren vorgestellt.

1.2.1 Zustand

Wie jedes komplexere Datenverarbeitungsystem, so hat auch ein Scheme–System einen Zustand. Der verteilt sich hier auf drei Funktionseinheiten: Speicher, Ausführungs– und Übersetzungsumgebungen.

Speicher. Der hier betrachtete Speicher besteht aus einer Reihe von Zellen, die jeweils genau einen Wert aufnehmen können. Ihr Inhalt kann sowohl gelesen als auch verändert werden. Jede einzelne Zelle hat eine eindeutige Adresse.

Im Laufe einer Berechnung werden kontinuierlich neue Zellen benötigt, um (Zwischen–)Ergebnisse aufzubewaren. In einem idealen System gäbe es unendlich viel Speicher. Dann ginge der Vorrat an leeren Zellen nie zur Neige. In einem relaen System ist aber alles endlich und daher der Speicher irgendwann voll. Um auch dann noch weiter rechnen zu können, werden nicht mehr erreichbare Zellen freigegeben und wiederverwendet. Die Suche nach solchen Zellen heißt *Garbage Collection*.

Ausführungsumgebungen. Bei der Auswertung eines Programmes benutzt der Interpreter Ausführungsumgebungen. Darin werden die Argumente von Funktionsaufrufen gesammelt und später im Rumpf der Funktion verfügbar gemacht. Die Werte selbst stehen allerdings im Speicher; die Umgebungen enthalten nur deren Adressen.

Der Aufbau einer Ausführungsumgebung wird durch die lexikalische Sichtbarkeit von Variablen festgelegt: für jede geschachtelte Funktion gibt es einen Rahmen, der wiederum die Adressen der Argumente enthält. Wird im Programm einer dieser Werte verwendet, so geschieht das mit Hilfe von lexikalischen Adressen. Das sind Paare von Indizes; der erste bestimmt den Rahmen, der zweite die Adresse darin.

Übersetzungsumgebungen. Übersetzungsumgebungen dienen dazu, aus den kontextabhängigen Symbolen der externen Repräsentation von Programmen die kontextunabhängige lexikalische Adressen der internen Programme zu machen oder die Symbole als Sonderformen zu erkennen und dadurch den Übersetzungsvorgang zu beeinflussen. Dazu wird ein assoziativer, durch Symbole indizierter Speicher benötigt. Daher gleicht der Aufbau dieser Umgebungen nur teilweise dem der Ausführungsumgebungen: die Rahmen sind hier keine Tupel von Adressen sondern Mengen von Bindungen.

1.2.2 Sprachen

Der Aufbau des hier vorgestellten Scheme–Systems orientiert sich an drei Sprachen. Dabei ist eine Sprache eine Menge, deren Elemente bestimmten Bildungsgesetzen genügen.

Externe Repräsentationen. Die erste ist die im Scheme–Report definierte *externe Repräsentation*. Sie eignet sich sowohl zur Darstellung von Scheme–Daten als auch von Scheme–Programmen. Da sie aus Zeichenketten besteht, wird sie von einem Scheme–System bei der Kommunikation mit seiner Umwelt benutzt. Für die genaue Spezifikation dieser Sprache sei hier auf den Scheme–Report [19], Kapitel 7 verwiesen.

Da die externe Repräsentation sozusagen die Normalform für Scheme–Daten und –Programme ist, wird sie auch in dieser Arbeit häufig verwendet. Sie ist dann an der Verwendung `dieser Schrift` zu erkennen.

Daten. Die Manipulation von Zeichenketten ist nicht besonders effizient. Deshalb gibt es die Sprache der *internen Daten*. Sie sollen direkt und effizient manipulierbar sein. So wird zum Beispiel aus einer Folge von Ziffern in der externen Form ein mit einem Präfix versehenes Element von $\mathbb{Z}$: aus `42` wird NUM:42. Es gibt allerdings auch Daten, die keine externe Repräsentation haben. Funktionen sind ein Beispiel dafür, aber auch zyklische, aus Paaren aufgebaute Strukturen.

Programme. Die dritte Sprache stellt Programme dar. Scheme–Programme können Datenkonstanten enthalten, sogenannte Literale. Die Sprache der Programme muß daher die der Daten umfassen. Darüber hinaus können aber auch programmspezifische Konstrukte dargestellt werden, zum Beispiel Variablen oder Funktionsdefinitionen.

An dieser Stelle sei noch angemerkt, dass der Begriff “Ausdruck” hier synonym zu “Scheme–Programm” benutzt wird. Das entspricht nicht ganz den Bezeichnungen im Scheme–Report. Dort bestehen Programme aus Definitionen *oder* Ausdrücken. Allerdings stimmt die Bezeichnung wieder, wenn die hier benutzte interne Form der Programme betrachtet wird – und das ist in dieser Arbeit in der Regel der Fall. Definitionen werden hier vom Übersetzer behandelt und in Zuweisungen transformiert. Und das sind Ausdrücke.

1.2.3 Übersetzer

Daten werden in ihrer externen Form eingelesen und in ihre interne Form übersetzt. Von da aus können sie weiter in Programme transformiert oder auch wieder ausgegeben werden. Für jede dieser drei Umwandlungen gibt es einen Übersetzer.

1.2.4 Interpreter

Die Auswertung oder Interpretation eines Programmes geschieht schrittweise. Dabei wird jeweils der Inhalt einiger Register verändert. Neben einer Ausführungsumgebung und einem Programm ist darin auch eine Fortsetzung enthalten, die angibt, was mit dem Wert des Programmes zu tun ist.

Der Interpreter kennt dazu die Semantik der Programme in Form einer Menge von Regeln. Jede dieser Regeln besteht aus einem Muster für die Registerinhalte des Interpreters und einer Vorschrift, wie dieser zu verändern ist.

1.3 Ablauf

1.3.1 Start

Nach dem Starten des Scheme–Systems meldet es sich beim Benutzer mit einer Begrüßungsmeldung und der Eingabeaufforderung. Dies wird aber schon von einem Scheme–Programm geleistet.

Dadurch testet sich die Implementation gewissermaßen selbst und kann immer als erste von Verbesserungen profitieren. Das Scheme–System muß in der Implementations–Sprache nur soweit implementiert werden, dass es einen einfachen Ausdruck auswerten kann. An der Stelle steht dann natürlich noch nicht die ganze Funktionalität eines R^5RS–konformen Schemes zur Verfügung.

Dennoch ist eine gewisse Basisfunktionalität vorhanden. Unter anderem zum Beispiel die bereits erwähnten `read`– und `write`–Funktionen um Eingaben entgegennehmen und Ergebnisse ausgeben zu können. Allerdings ist `read` nicht die Funktion. Diese Sprechweise wird nur zur Abkürzung benutzt. Korrekt müßte es heißen: Die Funktion, die in der Speicherzelle enthalten ist, auf die die lexikalische Adresse verweist, die in der initialen globalen Umgebung an die Variable gebunden ist, die durch das Symbol dargestellt wird, dessen externe Repräsentation `read` ist. Die Abkürzung ist also sehr sinnvoll.

Allerdings lehrt dieser kurze Exkurs in die korrekte Sprechweise folgendes: Damit die `read`–Funktion verfügbar ist, muß der Speicher existieren und mit Werten gefüllt sein. Die globale Umgebung, bestehend aus Übersetzungs– und Ausführungsumgebung, muß ebenfalls erzeugt und befüllt worden sein. Um die Funktion aufrufen zu können muß der Interpreter bereit stehen und die notwendigen Aktionen koordinieren. Damit die Funktion arbeiten kann, muß der extern→intern–Daten–Übersetzer verfügbar sein.

Um die Verfügbarkeit des Interpreters und der Übersetzer und um den Aufruf einer Startfunktion kümmert sich in der Regel die Laufzeit–Umgebung der Implementations–Sprache oder ein Betriebssystem. Die speziellen Datenstrukturen der Implementation müssen jedoch von ihr selbst erzeugt und gefüllt werden. Dieser Vorgang wird *bootstrapping* oder *booten* genannt.

In der Startfunktion werden zunächst der Speicher[1] und die Umgebungen als leere Datenstrukturen angelegt. Danach wird eine Tabelle, die eingebaute (sogenannte primitive) Funktionen und deren Namen enthält, ausgelesen. Jede Funktion wird in eine Speicherzelle, deren Adresse in die Ausführungsumgebung und schließlich die entsprechende lexikalische Adresse in die Übersetzungsumgebung eingetragen.

Eine weitere Tabelle enthält die Sonderformen, die nur in der Übersetzungsumgebung ihre Spuren hinterlassen. Ihre Namen werden nicht mit lexikalischen Adressen, sondern mit speziellen Steuerzeichen verknüpft, die den Daten$\rightarrow$Programme–Übersetzer beeinflussen.

Damit ist das Scheme–System in der Lage, einfache Scheme–Programme zu übersetzen und auszuführen. Wie bereits erwähnt kennt es aber noch nicht alle im Scheme–Report definierten Funktionen. Viele davon lassen sich einfacher in Scheme als in der Implementations–Sprache ausdrücken.

Daher wird das System nun ein erstes Mal aktiviert. Ein als Zeichenkette (also in externer Repräsentation) in der Implementation enthaltenes Scheme–Programm wird zunächst in Daten, dann in ein Programm übersetzt und schließlich vom Interpreter ausgeführt. Dabei werden zum Beispiel einige der *library procedure*s des Scheme–Reports zu der initialen Umgebung hinzufügt. Das Ergebnis dieser Auswertung wird ignoriert.

Nun ist das Scheme–System bereit, beliebige Ausdrücke auszuwerten. Allerdings steht diese Funktionalität zunächst nur auf der Ebene der Implementation zur Verfügung.

Um das System interaktiv zu machen, muß eine Read–Eval–Print–Schleife gestartet werden. Das ist mit Hilfe der `read`–, `eval`– und `write`–Funktionen auch in Scheme möglich. Ein entsprechendes Programm ist Teil der Implementation. Aufgabe der Startfunktion ist es nun nur noch, diese Zeichenkette dem Scheme–System zur Auswertung zu übergeben. Da die Schleife im Programm steckt, wird nach Beendigung der Auswertung nur das Resultat und eine Abschiedsmeldung ausgegeben. Damit ist die Arbeit des Scheme–

[1]In der später vorgestellten Java–Implementation benutzt das Scheme–System die Speicherverwaltung des Java–Systems mit. Daher ist deren Initialisierung bereits vor dem Aufruf der Startfunktion abgeschlossen.

Systems beendet.

1.3.2 Syntax–Prüfung und Übersetzung

Der erste Schritt einer jeden Übersetzung ist die Syntax–Prüfung. Da dabei die Struktur der Eingabe erkannt werden muß, ist es einfach, parallel dazu eine interne Repräsentation dieser Struktur zu erzeugen.

Daten. Im Falle des extern→intern–Daten–Übersetzers ist das auch schon alles. Seine Zielsprache ist eben diese interne Repräsentation. Er enthält auch keine Zustandsinformation, die einzelne Übersetzungen überdauert: Die Syntax der Daten ist kontextfrei, denn ein externes Datum erzeugt immer dasselbe interne, egal wo es in einer Eingabe vorkommt. Dennoch verändert der Übersetzer den Zustand des Scheme–Systems: Beim Übersetzen zusammengesetzter Strukturen trägt er die Teile im Speicher ein.

Programme. Im Gegensatz zur Syntax der Daten ist die der Programme nicht kontextfrei. Symbole, die in Programmen *Bezeichner* genannt werden, können gebunden oder frei sein. Sind sie an Syntax gebunden, so werden sie *Sonderformen* genannt, sind sie an lexikalische Adressen gebunden, so heißen sie *Variablen*.

Was an einer bestimmten Stelle eines Programmes an einen Bezeichner gebunden oder ob er frei ist, hängt von seinem Kontext im Programm ab. Die daraus benötigten Informationen verwaltet der Übersetzer in den Übersetzungsumgebungen.

Mit Hilfe der jeweils aktuellen Umgebung kann der Übersetzer jedem Bezeichner in einem Programm entweder seine korrekte Bedeutung zuordnen oder einen Fehler melden. Dazu werden die Rahmen von innen nach außen durchsucht. Die erste gefundene passende Bindung liefert das gewünschte Resultat. Gibt es keine, so liegt ein Syntax–Fehler vor.

Die einfachsten Programme sind Zahlen, Zeichen, Zeichenketten oder boole'sche Werte. Daten dieser Typen werden zu Literalen.

Komplexere Ausdrücke bestehen aus Listen. Wie eine solche Liste auszuwerten ist, bestimmt ihr erstes Element, ihr *Kopf*. Normalerweise stellen sie den Aufruf einer Funktion dar: Der Kopf ist der Operator, die restlichen Elemente sind die Operanden. Der mathematische Ausdruck $f(x_1, x_2, \ldots, x_n)$ würde in Scheme also als `(`$f\ \ x_1\ \ x_2\ \ \ldots\ \ x_n$`)` dargestellt.

Enthält der Kopf jedoch eine Sonderform, so tritt eine andere Übersetzungsregel in Kraft. Welche das ist, hängt von der Sonderform ab.

Bei der Übersetzung eines `lambda`–Ausdrucks[2] wird vorübergehend die Übersetzungsumgebung verändert. Zunächst wird ein neuer Rahmen mit den Bindungen der formalen Parameter an lexikalische Adressen hinzugefügt. In der so entstandenen erweiterten Umgebung wird der Rumpf übersetzt.

Die Übersetzung eines `define`–Ausdrucks manipuliert dagegen nur den aktuellen Bindungsrahmen. Falls der im Ausdruck angegebene Bezeichner darin noch nicht gebunden ist, wird er mit einer neuen lexikalischen Adresse verknüpft.

Die anderen, dem Übersetzer bekannten Sonderformen verändern dessen Zustand nicht. Im einzelnen sind das:

- `quote`, um beliebige Literale – also beispielsweise auch Symbole oder Listen – in Programme einfügen zu können,
- `set!`, um den Inhalt bereits definierter Variablen zu ändern,
- `begin`, um eine Sequenz von Ausdrücken in einer definierten Reihenfolge auszuwerten und
- `if`, um durch einen berechneten Wert den Kontrollfluß steuern zu können.

Komplexere Sonderformen wie zum Beispiel `let` können aus den angegebenen aufgebaut werden.

Nach der Syntax–Prüfung und Übersetzung bestehen die Programme aus Literalen, lexikalische Adressen, normalen Funktionsaufrufen und den angegebenen Sonderformen ohne `define`. Die Art und Anzahl der Parameter der Sonderformen ist dann ebenfalls überprüft und korrekt.

1.3.3 Interpretation von Programmen

Nach der Übersetzung sind die Programme syntaktisch korrekt. Aufgabe des Interpreters ist es nun, sie entsprechend der im Scheme–Report angegebenen Semantik auszuwerten und den dabei errechneten Wert zurückzuliefern.

Diese Auswertung geschieht schrittweise. Der Interpreter enthält dazu mehrere Register. Der Inhalt des Programm–Registers bestimmt primär die in

[2] Das ist ein Listenausdruck mit der Sonderform `lambda` als Kopf. Da der die Semantik bestimmt, wird der ganze Ausdruck nach ihm benannt.

einem Schritt auszuführende Aktion. Unter anderem ist das in der Regel auch die Aktualisierung des Programm–Registers selbst.

Um eine lexikalischen Adresse im Programm–Register auszuwerten, wird der Inhalt eines weiteren Registers benötigt: die Ausführungsumgebung. Der Inhalt der dadurch bestimmten Speicherzelle wird als Literal in das Programm–Register eingetragen.

So oder anders erzeugte Literale im Programm–Register signalisieren das Vorliegen eines (Zwischen– oder End–)Resultats: Ein (Teil–)Ausdruck wurde vollständig ausgewertet. In diesem Falle bestimmt der Inhalt eines dritten Registers die auszuführenden Aktion. Enthält es den vor dem Start der Auswertung darin eingetragenen Wert, so hat der Interpreter seinen Endzustand erreicht und hält an. Ein anderer möglicher Inhalt ist eine *Fortsetzung*.

Fortsetzungen stellen den noch zu erledigenden Teil einer für Zwischenberechnungen unterbrochenen Auswertung dar. Daher müssen sie alle Registerinhalte speichern die für die Wiederaufnahme der unterbrochenen Auswertung nötig sind. Im einzelnen sind das die Inhalte des Ausführungsumgebungs– und des Fortsetzungs–Registers. Damit wird ein als linear verkettete Liste implementierter Fortsetzungs–Stack gebildet.

Liegen also ein Literal und eine Fortsetzung in den entsprechenden Registern vor, so werden zunächst die gespeicherten Registerinhalte zurückgeschrieben. Im Anschluß daran wird die von der Fortsetzung bestimmte unterbrochene Auswertung wieder aufgenommen und kann den im Literal enthaltenen Wert verarbeiten.

Es gibt Fortsetzungen für `if`–, `set!`– und `begin`–Ausdrücke. Zwei weitere dienen der Argumentauswertung und dem Funktionsaufruf.

1.3.4 Funktionen

Die Aufgabe des Interpreters ist in erster Linie die Steuerung des Kontrollflusses. Für die eigentliche Datenverarbeitung sind im Scheme–System die Funktionen zuständig.

Primitive Funktionen. Grundlegend für jede Berechnung sind die primitiven Funktionen. Sie sind Teil des Scheme–Systems und sind die notwendige Basis zur Lösung komplexerer Aufgaben. Sie ermöglichen zum Beispiel die Konstruktion von Listen und Vektoren, den Zugriff auf deren Elemente oder arithmetische Operationen. Einige interagieren auch mit dem Interpreter selbst.

Fortsetzungen. Eine Eigenschaft von Fortsetzungen ist es, einen Wert entgegenzunehmen. Das haben sie mit einparametrigen Funktionen gemeinsam. Scheme erlaubt es nun, Fortsetzungen auch wirklich wie solche Funktionen zu benutzen. Erzeugt werden diese Fortsetzungs–Funktionen mit Hilfe einer primitiven Funktion: `call-with-current-continuation`. Ihr Aufruf mit genau einem Argument führt zum Aufruf der enthaltenen Fortsetzung mit diesem Argument – mit allen Folgen, die das auf den Kontrollfluß hat: die eigentlich aktuelle Fortsetzung wird ignoriert und vergessen.

Das ermöglicht zum Beispiel die Beendung einer Berechnung analog zum Werfen einer Ausnahme in Java oder C++. Darüber hinaus kann aber auch eine so unterbrochene Berechnung später wieder aufgenommen werden, wenn die entsprechende Fortsetzung aufbewahrt wurde. Eine andere Anwendung ist die mehrfache Rückkehr von einem Funktionsaufruf um zB nichtdeterministische Berechnungen zu simulieren ([1], Abschnitt 4.3).

Closures. Die Auswertung von `lambda`–Ausdrücken produziert ebenfalls Funktionen. Die enthalten zunächst natürlich den übersetzten Rumpf. Darüber hinaus speichern sie noch den Inhalt des Ausführungsumgebungs–Registers, die Anzahl der im Rumpf definierten Variablen und die Anzahl der Parameter.

Wird eine solche Funktion aufgerufen, so wird zunächst die darin enthaltene Umgebung um einen neuen Rahmen in der ebenfalls gespeicherten Größe ergänzt. Der Rahmen wird mit neuen, bisher nicht benutzten Adressen gefüllt. Falls der Aufruf Argumente hatte, werden diese Werte in die ersten der neuen Speicherzellen eingetragen.

Die so neu erzeugte Ausführungsumgebung und der übersetzte Rumpf werden dann in die entsprechenden Register des Interpreters eingetragen. Damit ist der Aufruf auch schon abgeschlossen. Das normale Voranschreiten der Berechnung erledigt alles weitere.

Insbesondere werden keine Rückkehradressen gespeichert. Deren Aufgabe wird von den Fortsetzungen übernommen. Die werden bei einem Funktionsaufruf jedoch nicht verändert.

1.4 Ein Beispiel

Nachdem die Maschine erzeugt und gestartet wurde, wird der folgende Text eingelesen:

```
(define a 1)
```

Diese Zeichenfolge wird vom extern→intern–Daten–Übersetzer konsumiert. Dabei erzeugt er den Wert PAIR:$\langle\alpha_1, \alpha_2\rangle$ und füllt sechs Speicherzellen wie folgt mit Werten:

$$\begin{array}{rcl}
\sigma(\alpha_1) & = & \text{SYM:}\texttt{define} \\
\sigma(\alpha_2) & = & \text{PAIR:}\langle\alpha_3, \alpha_4\rangle \\
\sigma(\alpha_3) & = & \text{SYM:}\texttt{a} \\
\sigma(\alpha_4) & = & \text{PAIR:}\langle\alpha_5, \alpha_6\rangle \\
\sigma(\alpha_5) & = & \text{NUM:}1 \\
\sigma(\alpha_6) & = & \text{EMPTY}
\end{array}$$

Eine Syntax–Prüfung verlangt offenbar einigen Aufwand. Um den Typ des ersten Parameters festzustellen, müssen zwei Speicherzellen gelesen werden. Um die korrekte Länge der Parameter–Liste zu überprüfen noch zwei weitere. Es ist daher sehr sinnvoll, das *vor* der Interpretation zu erledigen.

Genau das leistet der Daten→Programme–Übersetzer: Er stellt zunächst fest, dass ein Paar vorliegt. Dessen erstes Element untersucht er näher. Da es ein Symbol ist, zieht er die Übersetzungsumgebung zu Rate und erfährt, dass es eine Sonderform ist. Es tritt die spezielle Übersetzungsregel für `define`–Ausdrücke in Kraft. Sie verlangt zwei weitere Elemente in einer Liste, das erste der beiden muß ein Symbol sein. Da die Maschine neu erzeugt wurde, ist das Symbol `a` noch nicht gebunden. Es wird also zum aktuellen Rahmen der Übersetzungsumgebung hinzugefügt. Das letzte Listenelement wird in der ergänzten Umgebung übersetzt. Als Zahl wird es zu einem Literal. Das Resultat ist schließlich das Programm

$$\textbf{assign:}\langle\textbf{ptr:}\langle 0, 107, \texttt{a}\rangle, \text{NUM:}1\rangle,$$

wobei 107 der Index der neuen Variablen im Rahmen der globalen Ausführungsumgebung ist. Zusätzlich dazu hat sich der Zustand des Übersetzers geändert, er kennt jetzt diese neue Variable namens `a`, dh im entsprechenden Rahmen der Übersetzungsumgebung gibt es jetzt das Element $\texttt{a} \mapsto \textbf{ptr:}\langle 0, 107, \texttt{a}\rangle$. Nun wird das Programm zur Ausführung an den Interpreter übergeben, also in dessen Programm–Register eingetragen.

Schritt 1: Der Präfix **assign:** bestimmt die auszuführende Aktion. Hier muß zuerst der Wert des Teilausdrucks `1` beziehungsweise des Teilprogramms NUM:1 berechnet werden. Also wird eine Fortsetzung erzeugt. Sie speichert, zusätzlich zu den Registerinhalten, die lexikalische

Adresse der zu setzenden Variablen. Dann wird das Teilprogramm im Programm–Register abgelegt.

Schritt 2: Da sich im Programm–Register ein Literal befindet, wird nun sofort die eben erzeugte Fortsetzung aufgerufen. Die Register werden – in diesem Falle unnötigerweise – restauriert. Danach veranlaßt diese Fortsetzung das Schreiben des erhaltenen Resultats in die von der lexikalischen Adresse und der aktuellen Ausführungsumgebung bestimmten Speicherzelle. Da das Resultat einer solchen Zuweisung in Scheme nicht definiert ist, kann zum Beispiel der Literal im Programm–Register verbleiben.

Schritt 3 und weiter: Die Auswertung des Beispiel–Programms ist beendet. Die Maschine fährt mit der Ausgabe des Ergebnisses und dem Einlesen weiterer Ausdrücke fort. Ihr Zustand hat sich bei obiger Auswertung allerdings geändert. Es gibt nun eine Variable namens `a`, die den Wert NUM:1 enthält.

2 Strukturen und Semantik vom Scheme

Nach dem allgemeinen Überblick über das System im vorigen Kapitel, folgt nun eine genaue Definition des Speichers, der Sprachen, des Interpreters und der Regeln die das Scheme–System ausmachen.

2.1 Speicher

Die Sprache Scheme enthält mehrere Mutations–Operatoren für zusammengesetzte Datenstrukturen und einen Zuweisungs–Operator für Variablen. Es ist sinnvoll, diese Operationen auf einen gemeinsamen Mechanismus zurückzuführen. Diesem Zwecke dient die Einführung des Speichers in Gestalt einer Menge von Addresse–Wert–Paaren als Teil der Maschine.

Die Menge **Adresse** wird hier nicht näher spezifiziert. Sie sollte hinreichend viele Elemente haben, für die eine Äquivalenzrelation existiert. Eine mögliche Wahl währe zum Beispiel $\mathbb{N}$, die natürlichen Zahlen, denn für theoretische Betrachtungen ist es sinnvoll, wenn diese Menge (und damit auch der Speicher) unendlich ist. Dann sind natürlich immer fast alle Speicherzellen leer.

Die Adressen $\alpha \in$ **Adresse**, die jeweils auf der linken Seite der Bindungen in σ stehen, sind immer paarweise verschiedenen. Daher kann jedes σ auch als

$$\begin{array}{rcl} \sigma \in \textbf{Speicher} & ::= & \{\alpha \mapsto v, \ldots\} \\ \alpha \in \textbf{Adresse} & & \end{array}$$

Abbildung 1: Speicher

partielle Funktion betrachtet werden. Deren Definitionsbereich ist dann eine endliche Teilmenge von **Adresse**: die Menge der vorkommenden Adressen.

Wird der Speicher im Zuge eines Zustandsübergangs geändert, so wird eine neue Menge σ' erzeugt, die diese Änderung widerspiegelt. Dabei wird die Schreibweise

$$\sigma' = \sigma[\alpha' \mapsto v']$$

verwendet, um den Wert v' an der Adresse α' in σ einzutragen. Für σ' gilt dann

$$\forall \alpha \in \textbf{Adresse} : \sigma'(\alpha) = \sigma[\alpha' \mapsto v'](\alpha) = \begin{cases} v' & \text{falls } \alpha = \alpha', \\ \sigma(\alpha) & \text{sonst.} \end{cases}$$

2.2 Daten

Wie Eingangs erwähnt, behandelt dieser Abschnitt die interne Darstellung der Daten im Interpreter. Im Scheme–System erfüllen sie wichtige Aufgaben:

Sie werden als Eingabe von Berechnungen konsumiert und stellen dabei vorhandene Informationen dar. Das kann zum Beispiel der gemessene Zustand eines physikalischen Systems, die aktuelle Uhrzeit oder ein Buchstabe sein.

Die im Scheme–Report definierte Struktur der Daten ist flexibel genug, um nahezu beliebige Informationen darstellen zu können. Sie besteht zum einen aus einer kleinen Menge atomarer Datentypen, die in Abschnitt 2.2.1 beschrieben werden, zum anderen bietet sie einige Möglichkeiten der Kombination von mehreren einzelnen Daten zu einem zusammengesetzten Datum. Ihnen ist Abschnitt 2.2.2 gewidmet. Abbildung 2 zeigt die formale Definition der hier benutzten internen Darstellung.

Die Menge **Zeichen** enthält alle Zeichen, die in der externen Darstellung von Daten vorkommen können. Für eine exakte Definition der Menge **Symbol** sei auf Abschnitt 7.1 des Scheme–Reports verwiesen. Informell kann man sagen, Symbole sind Zeichenfolgen, die sich als Name eignen und keine Leerzeichen oder Klammern enthalten.

$$
\begin{array}{rcll}
v & \in & \textbf{Wert} & ::= \ \textsc{true} \mid \textsc{false} \mid \textsc{empty} \\
 & & & \quad \mid \ \textsc{char}{:}c \mid \textsc{num}{:}z \mid \textsc{sym}{:}I \\
 & & & \quad \mid \ \textsc{pair}{:}\langle \alpha_0, \alpha_1 \rangle \mid \textsc{vec}{:}\langle \alpha, \ldots \rangle \mid \textsc{str}{:}\langle \alpha, \ldots \rangle \\
 & & & \quad \mid \ \textsc{unspecified} \mid \textsc{undefined} \\
 & & & \quad \mid \ f \\
f & \in & \textbf{Funktion} & ::= \ \textsc{primop}{:}\psi \mid \textsc{apply} \mid \textsc{callcc} \\
 & & & \quad \mid \ \textsc{escape}{:}\langle \alpha, \kappa \rangle \mid \textsc{closure}{:}\langle \alpha, m, n, E, \rho \rangle \\
c & \in & \textbf{Zeichen} & \\
z & \in & \mathbb{Z} & \\
I & \in & \textbf{Symbol} & \\
\psi & \in & \textbf{Wert}^* \to \textbf{Wert} &
\end{array}
$$

Abbildung 2: Syntax der Daten.

2.2.1 Einfache Daten

Daten der folgenden Typen sind die elementaren Bausteine, aus denen sich Daten in Scheme zusammensetzen. Sie sind unveränderbar, weshalb sie sich ohne weiteres kopieren lassen. Original und Kopie sind immer äquivalent, je nach Implementation sogar identisch.

Die in den folgenden Absätzen angegebenen Beispiele zeigen jeweils den Zusammenhang zwischen der externen und der internen Darstellung eines Datums des entsprechenden Typs.

Boole'sche Werte. Um in einem `if`–Ausdruck (2.3.5) die Alternative auszuwählen benutzt Scheme den Wert FALSE. Da jeder andere Wert die Folge auswählt, ist es zwar nicht notwendig aber eleganter auch einen Wert TRUE zu definieren und damit einen vollständigen Boole'schen Typ zu schaffen.

Beispiele:
`#t` → TRUE
`#f` → FALSE

Die leere Liste. Listen im Allgemeinen sind natürlich nicht atomar. Sie werden durch verkettete Paare dargestellt. Um das Ende einer solchen Kette zu signalisieren, wird ein spezieller Wert benutzt: EMPTY, die leere Liste.

Beispiel: `()` → EMPTY

Zeichen. Mit Hilfe der beiden Funktionen `read-char` und `write-char` kann ein Scheme–System zeichenweise mit dem es umgebenden System kommunizieren. So können auch solche externen Daten bearbeitet werden, die nicht der Daten–Syntax von Scheme entsprechen.

Dazu müssen die Elemente der Menge **Zeichen** auch als Werte im Scheme–System auftreten können. Zu diesem Zwecke werden sie mit dem Präfix CHAR: versehen.

Beispiele:
`#\a` → CHAR:`a`
`#\5` → CHAR:`5`

Zahlen. Um numerische Berechnungen durchführen zu können, definiert der Scheme–Report eine ganze Familie von Zahlentypen. In dieser Arbeit steht jedoch das Scheme–System selbst im Vordergrund und nicht dessen numerische Fähigkeiten. Daher werden hier nur mit dem Präfix NUM: versehene ganze Zahlen als Werte definiert.

Beispiele:
`123` → NUM:123
`-42` → NUM:-42

Symbole. Scheme–Programme werden zunächst als Daten gelesen. Daher müssen die darin vorkommenden Bezeichner auch als Wert darstellbar sein. Auch bei der Ausführung von Programmen ist es manchmal sinnvoll, Namen zu Datenstrukturen hinzuzufügen, um sie identifizieren zu können. Ihre interne Darstellung besteht aus der mit dem Präfix SYM: versehenen Zeichenkette.

Ein Beispiel zur Verwendung von Symbolen aus [1], Abschnitt 2.4.2: Die beiden äquivalenten Darstellungen komplexer Zahlen, rechtwinklig und polar, sollen in einem Programm nebeneinander verwendet werden und müssen daher zu unterscheiden sein. Als Unterscheidungsmerkmal wird eines von zwei Symbolen zur Datenstruktur hinzugefügt: `rect` oder `polar`.

Beispiele:
`+` → SYM:`+`
`long-symbol-with-hyphens` → SYM:`long-symbol-with-hyphens`

Funktionen. In Scheme als funktionaler Sprache sind Funktionen auch Werte. Um ein Scheme–System nutzen zu können, müssen einige Funktionen von Anfang an vorhanden sein. Mit den Zahlen, zum Beispiel, könnte man wenig anfangen, gäbe es keine arithmetischen Operation. Daher sind solche grundlegenden oder primitiven Funktionen in die Maschine eingebaut und als ψ verfügbar. Zu Werten werden die ψ mit dem Präfix PRIMOP:.

Die Werte APPLY und CALLCC stehen ebenfalls für eingebaute Funktionen. Da sie aber im Gegensatz zu den ψ noch Informationen aus dem Zustand des Interpreters (2.5.3) verwenden, benötigen sie eine Sonderbehandlung und bekommen eigene Werte.

Die als CALLCC dargestellte Funktion dient dazu, Fortsetzungen (2.5.2) in spezielle Werte "einzupacken". Diese Fortsetzungs–Funktionen werden dann als ESCAPE:$\langle\alpha, \kappa\rangle$ dargestellt.

APPLY erlaubt es, eine als erstes Argument übergebene Funktion auf die restlichen Argumente anzuwenden.

Zusätzlich zu den eingebauten und automatisch erzeugten Funktionen kann der Benutzer in Scheme auch selbst definierte verwenden. Die Definition geschieht mit Hilfe von `lambda`–Ausdrücken (2.3.4), deren Auswertung Funktionen der Form CLOSURE:$\langle\alpha, m, n, E, \rho\rangle$ erzeugt.

In den erzeugten Funktionen ist jeweils eine eindeutige, bisher nicht verwendete Adresse α enthalten. Das geschieht, um sie einfacher vergleichen zu können. So ist es nicht notwendig, Äquivalenzrelationen auf Fortsetzungen oder übersetzten Programmen zu definieren, denn für die Adressen wird ohnehin eine benötigt.

Die Funktionen haben keine definierte externe Darstellung, daher gibt es hier keine Beispiele.

Spezielles. UNSPECIFIED wird vom Scheme–Report in Situationen verwendet, in denen es einer Implementation freigestellt ist, welchen Wert sie verwenden darf. Das gilt zum Beispiel für das Resultat der `set!`–Funktion. Für einen Benutzer des Scheme–Systems bedeutet das, dass ein solcher Wert zu ignorieren ist.

UNDEFINED ist der Wert, der in neu erzeugte Speicherzellen eingetragen wird. Er sollte im laufenden Programm niemals auftauchen, da jede Speicherzelle vor einem Auslesen vom Programm geschrieben worden sein muß. Taucht er dennoch auf, so deutet das auf einen Fehler hin – möglicherweise sogar im Interpreter oder Übersetzer.

Beide Werte haben keine definierte externe Darstellung.

2.2.2 Zusammengesetzte Daten

Mit Werten der folgenden Typen lassen sich rekursiv beliebige Datenstrukturen aufbauen. Im Gegensatz zu atomaren Werten können zusammengesetzte Datenstrukturen verändert werden, da sie Adressen enthalten.

Paare. Der einfachste zusammengesetzte Wert ist das Paar. Es ist zusammen mit EMPTY Baustein von Listen und damit elementar im Scheme–System. Wie der Name schon sagt, enthält es immer genau zwei Teilwerte. Es wird in der internen Darstellung mit dem Präfix PAIR: versehen.

Erzeugt werden Paare mit der Konstruktor–Funktion `cons`. Auf die beiden enthaltenen Werte kann mit Hilfe der Selektor–Funktionen `car` und `cdr` zugegriffen werden. Die beiden Mutator–Funktionen `set-car!` und `set-cdr!` ermöglichen es, das Paar zu verändern.

Beispiel:
`(1)` $\rightarrow$ PAIR:$\langle \alpha_1, \alpha_2 \rangle$ mit $\sigma(\alpha_1) =$ NUM:1 und $\sigma(\alpha_2) =$ EMPTY.

Vektoren. In Listen müssen zum Zugriff auf ein bestimmtes Element zuerst alle davor liegenden Paare besucht werden. Für Algorithmen, bei denen das zu inakzeptablen Laufzeiten führen würde, bietet Scheme Vektoren. Jeder Zugriff geht in konstanter Zeit vonstatten.

Auch in Vektoren können beliebige Werte enthalten sein. Deren Anzahl wird bei der Erzeugung des Vektors festgelegt und kann danach nicht mehr verändert werden.

Da die Größe also a priori nicht festgelegt ist, benötigen sowohl der Konstruktor `make-vector` als auch der Selektor `vector-ref` und der Mutator `vector-set!` ein nicht–negatives, ganzzahliges Argument, das Größe beziehungsweise Index angibt.

Beispiel:
`#(#\a #\b)` $\rightarrow$ VEC:$\langle \alpha_1, \alpha_2 \rangle$ mit $\sigma(\alpha_1) =$ CHAR:`a` und $\sigma(\alpha_2) =$ CHAR:`b`

Zeichenketten. Eine Sonderform des Vektors ist die Zeichenkette. Sie kann – im Gegensatz zu Paaren und Vektoren – nur Zeichen enthalten. Ihre externe Darstellung ist einfach die Folge ihrer Elemente. Dadurch lassen sich Zeichenfolgen in Programmen erheblich kompakter angeben, als durch Vektoren, die Zeichen enthalten. Weiterhin erlauben sie die Ein– und Ausgabe beliebiger Texte in direkt lesbarer Form.

$$
\begin{array}{rclcl}
E & \in & \textbf{Programm} & ::= & v \\
 & & & | & A \\
 & & & | & \textbf{apply:}\langle E_{\text{op}}, E, \ldots \rangle \\
 & & & | & \textbf{lambda:}\langle m, n, E \rangle \\
 & & & | & \textbf{select:}\langle E, E^{+}, E^{-} \rangle \\
 & & & | & \textbf{assign:}\langle P, E \rangle \\
 & & & | & \textbf{seqence:}\langle E_1, E_2, E_3, \ldots \rangle \\
A & \in & \textbf{LexAdresse} & ::= & \textbf{ptr:}\langle m, n, I \rangle \\
m, n & \in & \mathbb{N}_0 & &
\end{array}
$$

Abbildung 3: Syntax der Zwischensprache.

Durch den festgelegten Typ ihrer Elemente können Zeichenketten effizienter implementiert werden als Vektoren.

Die Basisfunktionen für Zeichenketten entsprechen denen für Vektoren, wobei `vector` durch `string` ersetzt wird.

Beispiele:
`"ab"` $\rightarrow$ STR:$\langle \alpha_1, \alpha_2 \rangle$ mit $\sigma(\alpha_1) =$ CHAR:`a` und $\sigma(\alpha_2) =$ CHAR:`b`

2.3 Programme

Hier wird die Syntax der verwendeten Sprache für Programme und deren Beziehung ihrer externen Form vorgestellt. Abbildung 3 zeigt eine formale Definition der internen Form. Die darin vorkommenden Sprachelemente werden in den folgenden Abschnitten jeweils einzeln besprochen.

2.3.1 Literale

$\langle$*Boolean*$\rangle$
$\langle$*Zahl*$\rangle$
$\langle$*Zeichen*$\rangle$
$\langle$*Zeichenkette*$\rangle$
`'`$\langle$*Datum*$\rangle$
`(quote` $\langle$*Datum*$\rangle$`)`

Die einfachsten möglichen Programme in Scheme sind die Literale. Ihre Auswertung ergibt ohne jede weitere Berechnung immer den selben Wert. Listen, Symbole und Vektoren müssen durch Quotierung explizit als Literale gekennzeichnet werden. Werte anderer Datentypen werden es in Programmen automatisch.

Daten in interner Darstellung sind also eine Teilmenge der Zwischensprache und werden darin durch Programme der Form v dargestellt.

Beispiele:
`42` $\rightarrow$ NUM:42
`'42` $\rightarrow$ NUM:42
`'(a)` $\rightarrow$ PAIR:$\langle\alpha_1, \alpha_2\rangle$ mit $\sigma(\alpha_1) =$ SYM:`a` und $\sigma(\alpha_2) =$ EMPTY

2.3.2 Variablen

⟨*Symbol*⟩

Einige Arten von Scheme–Werten können nicht direkt Teil von Ausdrücken sein – zumindest dann nicht, wenn sie vom Scheme Parser erzeugt wurden. Dazu gehören insbesondere die eingebauten Funktionen, die vom Scheme Interpreter direkt ausgeführt werden. Um sie aber dennoch als Teile von Ausdrücken benutzen zu können, bekommen sie Namen: sie werden an Scheme–Symbole gebunden, die damit als *Variablen* fungieren.

Solche Bezeichner erleichtern es (menschlichen) Entwicklern, Programme zu schreiben und zu verstehen. Dazu sollten sie natürlich sinnvoll gewählt sein wie zum Beispiel `+` für eine Additionsfunktion oder `write` für eine Ausgabe–Funktion. Für (maschinelle) Interpreter sind sie als Symbole dagegen erheblich weniger nützlich. Da die Zwischensprache für letztere konzipiert wurde, werden die Symbole zu lexikalischen Adressen der Form **ptr:**$\langle m, n, I\rangle$, welche die Informationen enthalten, die der Interpreter benötigt. Das Symbol ist Teil der lexikalischen Adresse, um die Beziehung zum ursprünglichen Scheme–Programm sichtbar zu machen. Benötigt wird es dort nicht mehr.

Die lexikalische Bindung der Sprache Scheme ermöglicht diese Übersetzung. Schon bei der Syntax–Prüfung steht fest, an welcher Stelle der Ausführungsumgebung die Adresse des an ein Symbol gebundenen Wertes zu finden ist. Die entsprechende Information ist in den beiden Indizes codiert, die im Zeiger enthalten sind ([1], Abschnitt 5.5.6).

Beispiele[3]:
`+` $\rightarrow$ **ptr:**$\langle 0, 34, \texttt{+}\rangle$
`write` $\rightarrow$ **ptr:**$\langle 0, 16, \texttt{write}\rangle$

[3] Die angegebenen Indizes sind der Implementation entnommen.

2.3.3 Aufruf von Funktionen

`(`$\langle$*Operator*$\rangle$ $\langle$*Operand*$\rangle$ ...`)`

Nach dem direkten und indirekten Zugriff auf vorhandene Daten, sollen sie nun kombiniert und so zum Beispiel eine der schon angesprochenen eingebauten Funktionen aufgerufen werden. Der in der Literatur für diese Operation häufig verwendete Name 'Kombination' leitet sich von ihrer Darstellung im Programm ab: Als Liste von aufeinander folgenden – kombinierten – Teilausdrücken, die von Klammern umschlossen sind. Das erste Element heißt *Operator*, die folgenden *Operanden*. Eine leere Operanden–Liste ist möglich, ein Operator muß dagegen immer vorhanden sein. Die leere Liste ist also keine Kombination. Weiterhin darf der Operator keine Sonderform sein. Listen mit Sonderformen an erster Stelle werden in den folgenden Abschnitten vorgestellt.

Der Wert einer Kombination wird ermittelt, indem zunächst alle Teilausdrücke ausgewertet werden und dann die durch den Operator bestimmte Funktion auf die Werte der Operanden, die *Argumente* angewandt wird.

Zur Darstellung in der Zwischensprache werden die übersetzten Teilausdrücke in einem Tupel zusammengefaßt und mit dem Präfix **apply:** versehen.

Beispiel:
`(+ 1 2)` $\rightarrow$ **apply:**$\langle$**ptr:**$\langle 0, 34, +\rangle, \text{NUM}{:}1, \text{NUM}{:}2\rangle$

2.3.4 Erzeugung von Funktionen

`(lambda` $\langle$*formale Parameter*$\rangle$ $\langle$*Ausdruck 1*$\rangle$ $\langle$*Ausdruck 2*$\rangle$ ...`)`

Mit Hilfe der Kombinationen können beliebig komplexe Ausdrücke konstruiert werden. Aber es fehlt noch eine Methode, um die wachsende Komplexität wieder zu strukturieren, um wiederkehrende Berechnungen nur einmal implementieren zu müssen. Es fehlt noch die Möglichkeit, Algorithmen zu einer *Abstraktion* zusammenfassen zu können. In der Mathematik dienen Funktionen diesem Zweck. So ist es auch in Scheme. Funktionen sind – wie auch Zahlen oder Zeichen – Daten und können in Variablen gespeichert werden. So können Funktionen dann auch mit Namen versehen werden. Der Algorithmus, den sie verbergen, wird ausgeführt, wenn sie als Wert des Operators in einer Kombination auftreten.

Die Funktionen selbst sind Werte und es wurde bereits erwähnt, dass sie keine externe Darstellung haben. Sie können also auch nicht direkt Teil von Programmen sein. Was es hier also zu betrachten gilt, sind Programme, deren

Ausführung Funktionen produziert: die `lambda`–Ausdrücke. Wie Kombinationen werden auch sie durch Listen dargestellt. Die an der Operator–Position stehende Sonderform `lambda` kennzeichnet sie aber als Ausdruck mit eigenen Auswertungsregeln.

Auf das einleitende Symbol folgt als zweites Element eine Liste von Symbolen die als *formale Parameter* dienen. Die restlichen Elemente – der *Rumpf* des `lambda`–Ausdrucks – sind eine Folge von Ausdrücken, welche die oben erwähnte vorbereitete aber noch nicht auszuführende Berechnung darstellt.

Bei der Auswertung des `lambda`–Ausdrucks wird der Rumpf nicht ausgewertet sondern eine Funktion erzeugt, die ihn, zusammen mit der aktuelle Umgebung enthält.

Wird eine so erzeugte Funktion auf eine Argument–Liste angewandt, so wird die gespeicherte Umgebung um einen Rahmen mit den Bindungen der in der Liste enthaltenen Werte an die formalen Parameter ergänzt und die Ausdrücke des Rumpfes in der so entstandenen Umgebung ausgewertet.

Da die Bezeichner nur bei der Übersetzung in die Zwischensprache eine Rolle spielen und dabei durch lexikalische Adressen ersetzt werden, ist im übersetzten `lambda`–Ausdruck nur die Anzahl der Parameter und lokalen Variablen enthalten. Der Rumpf wird nicht als Folge von Teilprogrammen gespeichert, ggf. wird bei der Übersetzung auf die in 2.3.7 vorgestellte Sequenz zurückgegriffen, um die Folge in ein Programm zu übersetzen.

Beispiel:
`(lambda (x y z) (+ x z))` $\rightarrow$

$$\textbf{lambda:}\langle 3, 0, \textbf{apply:}\langle \textbf{ptr:}\langle 0, 34, \texttt{+}\rangle, \textbf{ptr:}\langle 1, 0, \texttt{x}\rangle, \textbf{ptr:}\langle 1, 2, \texttt{z}\rangle\rangle\rangle$$

2.3.5 Bedingte Auswertung

`(if` $\langle$*Prädikat*$\rangle$ $\langle$*Folge*$\rangle$ $\langle$*Alternative*$\rangle$`)`

Mit den bisher vorgestellten Elementen der Zwischensprache ist nur die sofortige und die verzögerte Auswertung möglich. Es fehlt noch die bedingte Auswertung, dargestellt durch `if`-Ausdrücke.

Wird ein `if`-Ausdruck ausgewertet, so wird zuerst sein *Prädikat* ausgewertet. Das Ergebnis bestimmt, ob die *Folge* oder die *Alternative* benutzt wird, um den Wert des Gesamtausdrucks zu berechnen.

In der Zwischensprache werden die Teilausdrücke wieder zu einem Tupel von Teilprogrammen zusammengefasst.

Es ist möglich, `if` durch eine primitive Funktion `if-func` zu nachzubilden. Die

Folge und Alternative müssen dazu in parameterlose `lambda`–Ausdrücke verpackt werden. `if-func` werden dann das ausgewertete Prädikat und zwei Funktionen übergeben, von denen eine zurückgegeben wird. Die wird dann aufgerufen.

`(if a b c)` $\equiv$ `((if-func a (lambda () b) (lambda () c)))`

Der Nachteil dieses Vorgehens ist, dass dabei mehrere anonyme Funktionen erzeugt werden müssen. Dazu werden Umgebungen manipuliert, was unter Umständen aufwändig sein kann und in diesem Zusammenhang nicht notwendig ist. Im Sinne einer effizienteren Interpretation wurde hier deshalb die (geringfügig) höhere Komplexität des Interpreters in Kauf genommen.

Beispiel:
`(if 1 2 3)` $\rightarrow$ **select:**$\langle$NUM:1, NUM:2, NUM:3$\rangle$

2.3.6 Zuweisung

`(set!` $\langle$*Bezeichner*$\rangle$ $\langle$*Ausdruck*$\rangle$`)`
`(define` $\langle$*Bezeichner*$\rangle$ $\langle$*Ausdruck*$\rangle$`)`

Die in 2.3.4 betrachteten Funktionen haben – in Form der gespeicherten Umgebung – einen Zustand. Allerdings wird der bei ihrer Erzeugung festgelegt und kann mit den bisher vorgestellten Mitteln nicht verändert werden. Um aber Objekte durch Funktionen modellieren zu können, deren Verhalten von ihrer Vorgeschichte beeinflußt wird, gibt es in Scheme einen Zuweisungs–Operator. Mit ihm läßt sich der Inhalt einer Variablen ändern.

In der Zwischensprache enthält die Zuweisung die dem Bezeichner entsprechenden lexikalische Adresse und den übersetzten Ausdruck.

Beispiel:
`(set! + 17)` $\rightarrow$ **assign:**$\langle$**ptr:**$\langle 0, 34, +\rangle$, NUM:17$\rangle$

2.3.7 Sequenz

`(begin` $\langle$*Ausdruck 1*$\rangle$ $\langle$*Ausdruck 2*$\rangle$ `...)`

In Gestalt der Zuweisung gibt es also Ausdrücke, deren Wert nicht definiert ist und die nur wegen ihres Effektes auf den Zustand des Systems ausgewertet werden. Um diese als Teilausdrücke verwenden zu können und dennoch für den Gesamtausdruck einen definierten Wert zu erhalten, muß es einen Kontext geben, in dem der Wert eines Ausdrucks ignoriert wird.

Einen solchen Kontext bietet der `begin`-Ausdruck. Darin folgen auf das einleitende Symbol ein oder mehr Teilausdrücke, die in der angegebenen Rei-

$$
\begin{array}{lcllll}
P & \in & \textbf{ÜU} & ::= & \langle \Phi, \ldots \rangle \\
\Phi & \in & \textbf{ÜR} & ::= & \{B, \ldots\} \\
B & \in & \textbf{Bindung} & ::= & I \mapsto A \mid I \mapsto S \\
S & \in & \textbf{Syntax} & ::= & \textbf{quote} \mid \textbf{lambda} \mid \textbf{if} \mid \textbf{set} \mid \textbf{define} \mid \textbf{begin}
\end{array}
$$

Abbildung 4: Übersetzungsumgebungen und Syntax

henfolge ausgewertet werden. Der Wert des letzten ist dabei auch der Wert des Gesamtausdrucks, alle übrigen werden ignoriert.

Beispiel:
`(begin 1 2 3)` $\rightarrow$ **seqence:**$\langle$NUM:1, NUM:2, NUM:3$\rangle$

2.4 Daten→Programme–Übersetzer

2.4.1 Übersetzungsumgebungen

In Scheme können Symbole sowohl Variablen als auch Syntax darstellen. Sie werden dazu an eine lexikalische Adresse A oder ein Übersetzer–Steuerzeichen S gebunden. Solche Bindungen eines Symbols I werden als $I \mapsto A$ oder $I \mapsto S$ dargestellt.

Mehrere solcher Bindungen werden in einem Rahmen Φ zusammengefaßt. Dabei darf jedes Symbol höchstens einmal vorkommen.

Es ist allerdings erlaubt, dass in diesen Parameter–Listen Symbole vorkommen, die weiter außen bereits gebunden sind. Daher besteht eine vollständige Überstzungsumgebung P[4] aus einem Tupel mit ggf mehreren Rahmen.

So hat auch ein mehrfach vorkommendes Symbol in einer Übersetzungsumgebung eine eindeutige Bedeutung: Die erste, beim durchsuchen der Liste gefundene Bindung. Formal läßt sich das so beschreiben:

$$
P(I) := \Phi_m(I), \text{ falls } \begin{array}{l} P = \langle \Phi_1, \ldots, \Phi_m, \Phi_{m+1}, \ldots, \Phi_n \rangle, \\ \Phi_k(I) \text{ nicht definiert für } m < k \leq n \text{ und} \\ \Phi_m \text{ enthält eine Bindung für } I. \end{array}
$$

Dabei wird eine Menge von Bindungen als partielle Funktion betrachtet, dh $M(k) = v$, falls $k \mapsto v \in M$.

[4] Das ist ein großes Rho, analog zu dem ρ der Rahmen von Ausführungsumgebungen.

Die formale Definition der beteiligten Mengen ist in Abbildung 4 wiedergegeben. Dort wird auch die Menge **Syntax** definiert. Deren Elemente werden bei der Übersetzung benutzt, um anzuzeigen, als was eine Liste zu behandeln ist.

2.4.2 Übersetzungsfunktion

Die Übersetzungsfunktion

$$\tau : \begin{array}{rcl} \mathbf{Wert} \times \mathbf{\ddot{U}U} & \rightarrow & \mathbf{Programm} \times \mathbf{\ddot{U}U} \\ v, P & \mapsto & \langle E, P' \rangle \end{array}$$

transformiert mit Hilfe einer Übersetzungsumgebung einen Wert in ein Programm. Dabei wird überprüft, ob der Wert als syntaktisch korrektes Programm in Frage kommt. Nur dann kann die Transformation erfolgreich durchgeführt werden. Abbildung 5 zeigt eine formale Spezifikation der Funktion τ. Dabei wurden einige Details noch unterschlagen. So ist zum Beispiel nicht angegeben, wie die lexikalischen Adressen erzeugt werden.

Für die meisten Datentypen ist die Funktion τ sehr einfach. Für Zahlen, Zeichen, Zeichenketten, TRUE und FALSE ist sie die Identität.

Für die leere Liste und Vektoren ist sie nicht definiert, da Werte dieser Typen keine Ausdrücke sind. Für einige Typen (Funktionen, Ports, etc.) schreibt der Scheme–Report nicht vor, was aus ihnen wird, wenn sie in Ausdrücken auftauchen. Das liegt daran, dass sie nicht Teil der dort in Kapitel 7 definierten Sprache sind. In einem externen Programmtext können sie daher nicht vorkommen.

Bei Symbolen leistet τ das erste Mal etwas mehr Arbeit: Sie werden in der Übersetzungsumgebung gesucht. Dazu werden die darin enthalten Rahmen einer nach dem anderen nach einer Bindung des zu übersetzenden Symbols durchsucht. Die erste gefundene wird genauer betrachtet. Enthält sie eine lexikalische Adresse, so ist diese das Ergebnis der Übersetzung. Wird jedoch ein Übersetzer–Steuerzeichen oder überhaupt keine Bindung gefunden, so ist das ein Fehler und das Ergebnis ist undefiniert.

Ganz besonders komplex wird das Verhalten von τ, wenn das Wert–Argument eine Liste ist. Dann muß als erstes festgestellt werden, als was diese Liste zu übersetzen ist. Diese Information wird ihrem Kopf entnommen. Ist der kein Symbol oder eine Variable, so handelt es sich um eine Kombination, einen Funktionsaufruf. Alle Elemente der Liste werden übersetzt, die Resultate in einem Tupel gesammelt, mit dem entsprechenden Präfix versehen als Programm zurückgegeben.

$$\tau(v,P) \;:=\; \langle v,P\rangle \quad \text{falls } v \text{ ein boolescher Wert, eine Zahl, ein Zeichen oder einer Zeichenkette ist,}$$

$$\tau(\textsc{sym}{:}I,P) \;:=\; \langle A,P\rangle \quad \text{falls } P(I)=A,$$

$$\tau((\langle v_{op}\rangle\ \langle v_1\rangle\ \ldots\ \langle v_n\rangle),P) \;:=\; \begin{cases} \langle \textbf{apply:}\langle E_{op},E_1,\ldots,E_n\rangle,P_n\rangle \\ \quad \text{mit} \\ \qquad \begin{array}{rcl} \langle E_{op},P_0\rangle &=& \tau(v_{op},P) \\ \langle E_1,P_1\rangle &=& \tau(v_1,P_0) \\ &\vdots& \\ \langle E_n,P_n\rangle &=& \tau(v_n,P_{n-1}) \end{array} \\ \quad \text{falls } v_{op} \neq \textsc{sym}{:}\cdot \text{ oder} \\ \qquad v_{op} = \textsc{sym}{:}I \text{ und} \\ \qquad P(I) \in \textbf{Adresse}, \end{cases}$$

$$\tau((\langle \mathit{Symbol}\ I\rangle\ \langle v_1\rangle\ \ldots\ \langle v_n\rangle),P) \;:=\; \begin{cases} \langle v_1,P\rangle \\ \quad \text{falls } P(I)=\textbf{quote} \text{ und } n=1, \\ \langle \textbf{select:}\langle E_1,E_2,E_3\rangle,P_3\rangle \\ \quad \text{mit } E_i \text{ wie oben (setze } P_0 := P) \\ \quad \text{falls } P(I)=\textbf{if} \text{ und } n=3, \\ \langle E_1,P_1\rangle \\ \quad \text{falls } P(I)=\textbf{begin} \text{ und } n=1, \\ \langle \textbf{seqence:}\langle E_1,\ldots,E_n\rangle,P_n\rangle \\ \quad \text{falls } P(I)=\textbf{begin} \text{ und } n>1, \\ \langle \textbf{assign:}\langle A,E_2\rangle,P_2\rangle \\ \quad \text{mit } E_2 \text{ wie oben (setze } P_1 := P) \\ \quad \text{falls } P(I)=\textbf{set},\ n=2, \\ \qquad v_1 = \textsc{sym}{:}I' \text{ und } A = P(I'), \end{cases}$$

$$\begin{array}{l} \tau((\langle \mathit{Symbol}\ I\rangle\ \langle \mathit{Symbol}\ I'\rangle\ \langle v\rangle),\langle \Phi_1,\ldots,\Phi_{n-1},\Phi_n\rangle) := \langle \textbf{assign:}\langle A,E\rangle,P''\rangle \\ \quad \text{mit } \Phi'_n := \Phi_n[I' \mapsto A],\ P' := \langle \Phi_1,\ldots,\Phi_{n-1},\Phi'_n\rangle \text{ und } \langle E,P''\rangle := \tau(v,P') \\ \quad \text{falls } P(I) = \textbf{define} \end{array}$$

$$\begin{array}{l} \tau((\langle \mathit{Symbol}\ I\rangle\ (\langle \mathit{Symbol}\ I_1\rangle\ \ldots\ \langle \mathit{Symbol}\ I_m\rangle)\ \langle v\rangle),\langle \Phi_1,\ldots,\Phi_n\rangle) \\ \quad := \langle \textbf{lambda:}\langle m,\#\Phi'_{n+1}-m,E\rangle,\langle \Phi_1,\ldots,\Phi_n\rangle\rangle \\ \quad \text{mit } \Phi_{n+1} := \{I_1 \mapsto A_1,\ldots,I_m \mapsto A_m\},\ \langle E,\langle \Phi'_1,\ldots,\Phi'_{n+1}\rangle\rangle := \tau \end{array}$$

Abbildung 5: Übersetzung von Daten in Programme

In den verbleibenden Fällen ist das erste Element der Liste eine Sonderform, also ein an ein Übersetzersteuerzeichen gebundenes Symbol. Ist dieses Steuerzeigen **quote** und hat die Liste genau zwei Elemente, so werden weder der Wert noch die Umgebung verändert. Es wird ein Literal erzeugt.

Bei der Übersetzung von Listen mit **if**, **begin** und **set** wird ähnlich verfahren, wie bei der Kombination. τ wird rekursiv auf die restlichen Elemente der Liste angewandt und die Resultate in einem Tupel zusammengefaßt. Nur der Präfix, mit dem das Tupel versehen wird, ist jeweils ein anderer. Beispiele, wie solche übersetzten Ausdrücke aussehen, wurden bereits in Abschnitt 2.3 gegeben.

Die Übersetzung von Listen mit **define** verändert die Umgebung: Das neu definierte Symbol wird zum äußersten Rahmen der aktuellen Übersetzungsumgebung hinzugefügt. Ansonsten wird analog zu **set** vorgegangen.

Bei der Übersetzung von Listen, deren Kopf an **lambda** gebunden ist, wird die selbe Umgebung wieder zurückgegeben. Die Übersetzung des Rumpfes eines solchen `lambda`–Ausdrucks findet allerdings in einer erweiterten Umgebung statt. Dazu werden die formalen Parameter in einen neuen Rahmen eingetragen, der die Übersetzungsumgebung ergänzt. Wenn der Rumpf nun `define`–Ausdrücke enthält, so wird die Übersetzungsumgebung bei seiner Übersetzung noch weiter verändert. Danach enthält sie zusätzlich zu den Parametern auch noch lokale Variablen. Da später die Ausführungsumgebung für jede Variable eine Adresse enthalten muß, wird deren Anzahl ermittelt und in der internen Form des `lambda`–Ausdrucks abgespeichert.

2.5 Interpreter

Die bisher vorgestellten Strukturen werden erzeugt und danach nicht mehr verändert. Das entspricht der mathematischen Sicht, die keinen Zeitbegriff kennt. Die Maschine beziehungsweise deren Zustand soll sich jedoch im Laufe einer Auswertung ändern. Um das auch mathematisch zu fassen, wird eine Folge von Zuständen definiert. Die Zeit ist damit zum Index dieser Folge geworden. In der Realität, das heißt in einer Implementation, wird allerdings immer nur ein Glied der Folge gespeichert und vergessen, sobald ein neues erzeugt wurde.

Auf jeden Fall von Bedeutung sind aber die Elemente eines einzelnen Zustands. Sie werden in den folgenden Abschnitten betrachtet. In Abschnitt 2.6 werden dann die Regeln, nach denen aus einem gegebenen Zustand ein neuer erzeugt wird besprochen.

$$\begin{array}{rcrcl} \rho & \in & \mathbf{AU} & ::= & \langle \phi, \ldots \rangle \\ \phi & \in & \mathbf{AR} & ::= & \langle \alpha, \ldots \rangle \end{array}$$

Abbildung 6: Ausführungsumgebungen

2.5.1 Ausführungsumgebungen

Nur eine übersetzte Form eines `lambda`–Ausdrucks sollte für alle daraus erzeugten Funktionen verwendet werden können. Daher müssen die darin vorkommenden Symbole, also die formalen Parameter zu etwas werden, das nicht mit irgendwelchen Werten assoziiert ist. Wird eine solche Funktion dann aufgerufen, so müssen sich diese übersetzten formalen Parameter schnell und einfach an die konkreten Argumente binden lassen. Auch muß der Zugriff auf die Argumente effizient möglich sein.

Unter diesen Vorgaben wurden Schemes Symbole und Umgebungen zu lexikalischen Adressen und Ausführungsumgebungen.

Die lex. Adressen geben nur noch die Position des ursprünglichen Parameters in seiner Umgebung an. Die Ausführungsumgebungen enthalten keine Symbole mehr, sondern nur noch Adressen.

Wird eine `lambda`–Funktion aufgerufen, so werden die Argumente nur noch in Speicherzellen eingetragen. Deren Adressen werden – unter Beibehaltung der Reihenfolge – zu einem Tupel zusammengefaßt, das dann als neuer Rahmen die in der Funktion enthaltene Ausführungsumgebung ergänzt.

2.5.2 Fortsetzungen

Nur einfache Programme wie Literale, lex. Adressen und `lambda`–Ausdrücke lassen sich sofort auswerten. In der Regel sind Ausdrücke aber komplexer. Insbesondere enthalten sie Teilausdrücke, die zuerst ausgewertet werden müssen. Erst danach stehen deren Werte zur Verfügung. Die können dann zur Berechnung des Wertes des übergeordneten Ausdrucks verwendet werden.

Mit Hilfe von Fortsetzungen führt der Interpreter Buch über die noch zu erledigenden Aufgaben. Bei der Wiederaufnahme einer Berechnung muß sie aus der "zu erledigen"–Liste entfernt und den bei ihrer Unterbrechung aktuellen Zustand wieder hergestellt werden. So ist sichergestellt, dass zum Beispiel alle Elemente einer Sequenz in der selben Umgebung ausgewertet werden.

$$
\begin{array}{rcl}
\kappa \in \textbf{Fortsetzung} & ::= & \mathsf{call}{:}\langle\langle v, \ldots\rangle, \kappa'\rangle \\
& | & \mathsf{push}{:}\langle\langle E_{op}, E, \ldots\rangle, \langle v, \ldots\rangle, \kappa', \rho\rangle \\
& | & \mathsf{sequence}{:}\langle\langle E_1, E_2, \ldots\rangle, \kappa', \rho\rangle \\
& | & \mathsf{select}{:}\langle E^+, E^-, \kappa', \rho\rangle \\
& | & \mathsf{assign}{:}\langle A, \kappa', \rho\rangle \\
& | & \mathsf{stop}
\end{array}
$$

Abbildung 7: Syntax der Fortsetzungen

Zu diesem Zweck enthält jedes Tupel in den Elementen von **Fortsetzung** eine Ausführungsumgebung ρ. Die ebenfalls enthaltene Fortsetzung κ' ist die Verbindung zwischen den Elementen der "zu erledigen"–Liste.

Die verschiedenen Fortsetzungen, die im hier beschriebenen System vorkommen, sind in Abbildung 7 zu sehen. Im einzelnen sind das:

Funktionsaufruf. In Scheme muß der Kopf einer Kombination genauso wie alle weiteren Elemente ausgewertet werden. So ein Kopf kann natürlich ein beliebig komplexer Teilausdruck sein. Daher ist diese Fortsetzung notwendig. Sie bewahrt bereits ausgewertete Argumente auf und wartet auf die anzuwendende Funktion. Diese wird dann auf die gespeicherten Argumente angewandt. Was dabei genau geschieht, hängt von der Art der Funktion ab.

Diese Fortsetzung unterscheidet sich von den meisten anderen dadurch, dass sie keine Ausführungsumgebung enthält.

Argumentauswertung. Nur eine parameterlose Funktion kann sofort aufgerufen werden. In der Regel werden aber Argumente erwartet, die dann auch ausgewertet werden müssen. Dazu dient diese Fortsetzung. Sie speichert in einem Tupel die bereits fertigen Argumente und in einem weiteren die noch auszuwertenden Teilausdrücke.

Sequenz. Ein `begin`–Ausdruck wertet – wie eine Kombination – seine Argumente aus. Das geht bei mehr als einem Teilausdruck natürlich ebenfalls nicht in einem Schritt. Daher gibt es auch dafür eine Fortsetzung. Sie spei-

$$\textbf{Zustand} ::= \langle E, \kappa, \rho, \sigma \rangle$$

Abbildung 8: Syntax der Konfigurationen

chert aber nur ein Tupel mit den noch auszuwertenden Ausdrücken, da alle Ergebnisse außer dem letzten ignoriert werden.

Bedingte Ausführung. Die Aufgabe eines `if`–Ausdruckes ist es, genau zwei seiner drei Teilausdrücke auszuwerten. Auch dazu wird eine Fortsetzung benötigt. Sie enthält die beiden möglichen Kandidaten für den zweiten Auswertungsschritt.

Zuweisung. Bei einer Zuweisung muß der zuzuweisende Wert zuerst ausgerechnet werden. Das Ziel, eine Speicherzelle, muß solange zwischengespeichert werden. Das leistet eine Zuweisungs–Fortsetzung. Allerdings speichert sie eine *lexikalische* Adresse. Doch zusammen mit der ebenfalls enthaltenen Ausführungsumgebung ist die Zieladresse eindeutig bestimmt.

Stop. Wenn alle Teilausdrücke ausgewertet sind und alle Fortsetzungen aufgerufen wurden, muß der Interpreter ein Signal zum Anhalten bekommen. Diese Aufgabe fällt der letzten Fortsetzung aus Abbildung 7 zu. Sie enthält keine weiteren Daten, da sie weder einen Vorgänger hat, noch nach ihr etwas ausgewertet wird, das eine bestimmte Ausführungsumgebung benötigt.

2.5.3 Konfigurationen

Um die Aktionen des Interpreters einfacher beschreiben zu können, ist es sinnvoll, sie zu einem Tupel zusammen zufassen. Im Laufe einer Auswertung wird dann eine Folge solcher Tupel erzeugt. Jedes einzelne gibt eine Konfiguration, einen Zustand des Interpreters auf dem Weg zu einem Endergebnis an.

Im einzelnen sind in diesem Tupel der auszuwertende Ausdruck E, die auf das Ergebnis wartende Fortsetzung κ, die aktuelle Ausführungsumgebung ρ und der Speicher σ enthalten.

Die einzelnen Elemente des Tupels werden hier auch als Register bezeichnet.

$\langle$**ptr:**$\langle i, j, I\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle\sigma(\pi_j(\pi_i(\rho))), \kappa, \rho, \sigma\rangle$
$\langle$**lambda:**$\langle m, n, E\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle\textsc{closure}{:}\langle\alpha, m, n, E, \rho\rangle, \kappa, \rho, \sigma[\alpha \mapsto \textsc{unspecified}]\rangle$
mit einer bisher unbenutzten Adresse α.
$\langle$**select:**$\langle E, E^+, E^-\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle E, \mathsf{select}{:}\langle E^+, E^-, \kappa, \rho\rangle, \rho, \sigma\rangle$
$\langle$**assign:**$\langle A, E\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle E, \mathsf{assign}{:}\langle A, \kappa, \rho\rangle, \rho, \sigma\rangle$
$\langle$**apply:**$\langle E_{op}\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle E_{op}, \mathsf{call}{:}\langle\langle\rangle, \kappa\rangle, \rho, \sigma\rangle$
$\langle$**apply:**$\langle E_{op}, E_1, \ldots, E_{n-1}, E_n)\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle E_n, \mathsf{push}{:}\langle\langle E_{op}, E_1, \ldots, E_{n-1}\rangle, \langle\rangle, \kappa, \rho\rangle, \rho, \sigma\rangle$
$\langle$**seqence:**$\langle E_1, E_2, E_3, \ldots)\rangle, \kappa, \rho, \sigma\rangle$
$\quad\rightarrow\ \langle E_1, \mathsf{sequence}{:}\langle\langle E_2, E_3, \ldots\rangle, \kappa, \rho\rangle, \rho, \sigma\rangle$

Abbildung 9: Reduktionsregeln

2.6 Semantik

Hier werden die Regeln vorgestellt, welche die Zustandsübergänge der Maschine beschreiben. Dabei wird jeweils eine Vorher– und ein Nachher–Konfiguration angegeben.

2.6.1 Regeln für Programme

Diese in Abbildung 9 aufgeführten Regeln geben an, wie der aktuelle Ausdruck im Programm–Register zu reduzieren ist.

Lexikalische Adressen. Diese Ausdrücke können in einem Schritt ausgewertet werden. Die beiden Indizes i und j werden benutzt, um aus der Ausführungsumgebung ρ eine Adresse zu extrahieren.

Die so erhaltene Adresse wird benutzt, um aus dem Speicher σ einen Wert auszulesen. Der wird dann als Literal in das Programm–Register eingetragen.

`Lambda`–**Ausdrücke.** Sie erzeugen eine neue Funktion. Diese enthält zusätzlich zu den Informationen aus dem `lambda`–Ausdruck noch die aktuellen

Ausführungsumgebung, um die lexikalische Bindung zu ermöglichen und eine Adresse als eindeutiges Kennzeichen. Da die erzeugte Funktion ein Wert ist, wird sie ebenfalls als Literal ins Programm–Register eingetragen. Der Speicher wird ebenfalls verändert; die als Kennzeichen verwendete Adresse wird reserviert.

`If`–Ausdrücke. lassen sich nicht in einem Schritt ausführen. Zu ihrer Auswertung benötigt man zuerst den Wert des Test–Ausdrucks. Dessen Auswertung bereitet diese Regel vor. Der erste der drei Teilausdrücke wird in das Programm–Register eingetragen. Die beiden anderen warten in einer neuen Auswahl–Fortsetzung auf dessen Wert.

`Set!`– und `define`–Ausdrücke. Da auch hier ein Teilausdruck ausgewertet werden muß, wird eine neue Zuweisungs–Fortsetzung erzeugt. Sie enthält die lexikalische Adresse, an der später der Wert abgelegt werden soll. Der auszuwertende Teilausdruck wird in das Programm–Register eingetragen.

Kombinationen. Sie haben zwei Regeln. Die erste gilt für Kombinationen mit nur einem Element, also Funktionen ohne Argumente. Für sie wird sofort eine Aufruf–Fortsetzung erzeugt, die eine leere Argument–Liste enthält. Die Operanden mehrelementiger Kombinationen werden mit Hilfe einer oder mehrerer Schiebe–Fortsetzungen ausgewertet.

Die zweite der hier besprochenen Regeln beschreibt die Erzeugung der ersten dieser Fortsetzungen. Die Auswertung der Operanden geschieht dabei vom letzten zum ersten. Das ist legal, denn der Scheme–Report schreibt explizit vor, dass kein Scheme–Programm sich in diesem Zusammenhang auf eine bestimmte Reihenfolge verlassen darf.

`Begin`–Ausdrücke. oder Sequenzen bestehen immer aus mehr als einem Element. Die einelementige Sequenz ist zwar laut Scheme–Report legal, wird aber vom Übersetzer durch ihr Element ersetzt. Hier ist die Auswertungsreihenfolge natürlich festgelegt; das ist ja die Aufgabe einer Sequenz. Das erste Element wird also zur Ausführung in das Programm–Register eingetragen. Der Rest wird in eine Sequenz–Fortsetzung eingepackt und die dann in das Fortsetzungs–Register geschrieben.

$\langle v, \mathsf{stop}, \rho, \sigma\rangle$
$\quad\rightarrow$ Ende der Auswertung. Das Resultat ist v.
$\langle v, \mathsf{select}{:}\langle E^+, E^-, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle E, \kappa, \rho', \sigma\rangle$
$$\text{mit } E := \begin{cases} E^- & \text{falls } v = \text{FALSE}, \\ E^+ & \text{sonst.} \end{cases}$$
$\langle v, \mathsf{assign}{:}\langle \mathbf{ptr{:}}\langle i, j, I\rangle, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle \text{UNSPECIFIED}, \kappa, \rho', \sigma[\pi_j(\pi_i(\rho')) \mapsto v]\rangle$
$\langle v_n, \mathsf{push}{:}\langle\langle E_{op}, E_1, \ldots, E_{n-1}\rangle, \langle v_{n+1}, \ldots\rangle, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle E_{n-1}, \mathsf{push}{:}\langle\langle E_{op}, E_1, \ldots\rangle, \langle v_n, v_{n+1}, \ldots\rangle, \kappa, \rho'\rangle, \rho', \sigma\rangle$
$\langle v_1, \mathsf{push}{:}\langle\langle E_{op}\rangle, \langle v_2, \ldots\rangle, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle E_{op}, \mathsf{call}{:}\langle\langle v_1, v_2, \ldots\rangle, \kappa\rangle, \rho', \sigma\rangle$
$\langle v, \mathsf{sequence}{:}\langle\langle E_1, E_2, E_3, \ldots\rangle, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle E_1, \mathsf{sequence}{:}\langle\langle E_2, E_3, \ldots\rangle, \kappa, \rho'\rangle, \rho', \sigma\rangle$
$\langle v, \mathsf{sequence}{:}\langle\langle E\rangle, \kappa, \rho'\rangle, \rho, \sigma\rangle$
$\quad\rightarrow \langle E, \kappa, \rho', \sigma\rangle$

Abbildung 10: Fortsetzungsregeln

2.6.2 Regeln für Fortsetzungen

Diese in den Abbildungen 10 und 11 aufgeführten Regeln geben an, welche Aktionen der Inhalt des Fortsetzungs–Registers auslöst, wenn ein Literal im Programm–Register vorliegt. Der Literal ist das Resultat einer Berechnung, zeigt also deren Ende an. Die in der Fortsetzung gespeicherte, unterbrochene Berechnung kann dann wieder aufgenommen werden.

Ende der Auswertung. Die Konstante stop zeigt das Ende der Berechnung an. Sie ist eigentlich keine Fortsetzung, da die Maschine anhält. Auch speichert sie – im Gegensatz zu allen anderen Fortsetzungen – keine Registerinhalte.

Auswahl. Bei der Auswahl stehen zwei Teilprogramme zur Verfügung, von denen aber nur eines ausgewertet wird. Welches, das wird durch den der Fortsetzung übergebenen Wert bestimmt. Ist das die Konstante FALSE, so wird das zweite, sonst das erste Teilprogramm in das Programm–Register eingetragen. In dieser Regel manifestiert sich die Tatsache, dass in Schemes `if`–Ausdrücken jeder Wert, außer dem für Falsch, als Wahr gilt.

Zuweisung. Diese Fortsetzung ändert den an eine Variable gebundenen Wert. Dazu wird der Speicher des Scheme–Systems modifiziert. Der Rückgabewert dieser Operation ist im Scheme–Report nicht definiert. Daher wird hier der Wert UNSPECIFIED benutzt.

Argumentauswertung. Wie schon bei den Reduktionsregeln für Kombinationen, so gibt es auch hier zwei Regeln. Die erste tritt in Kraft, wenn alle Operanden ausgewertet sind. Die Argumente werden dann in einer Aufruf–Fortsetzung zusammengefaßt und der Operator in das Programm–Register eingetragen.

Sind noch weitere Operanden auszuwerten, so greift die zweite Regel. Eine neue Schiebe–Fortsetzung mit einer um ein Element längeren Argumente– und einer um ein Element kürzeren Operanden–Liste wird erzeugt. Der aus der Liste entfernte Operand wird in das Programm–Register eingetragen.

Sequenz. Auch hier gibt es wieder zwei Regeln. Enthält die Sequenz noch mehr als ein Element, so wird eine neue Fortsetzung für den Rest erzeugt. Das entfällt, wenn nur noch ein Ausdruck zur Auswertung ansteht.

In jedem Falle wird das erste Element der Sequenz in das Programm–Register eingetragen und damit als nächstes ausgewertet.

Aufruf. Für die Aufruf–Fortsetzung gibt es mehrere Regeln, jeweils eine für die verschiedenen Arten von Funktionen, die es in Scheme gibt. Sie sind in Abbildung 11 zusammengefaßt. Befindet sich im Programm–Register ein Wert, der von diesen Regeln nicht erfaßt wird, so liegt ein Fehler vor und die Auswertung muß abgebrochen werden.

Die erste Regel beschreibt, wie die eingebauten Funktionen aufgerufen werden. Die Funktion ψ wird auf die Argumente $v_1, \ldots, v_n$ angewandt. Dabei kann es natürlich zu Problemen kommen: n könnte nicht die Parameter–Anzahl von ψ sein. Oder eines der v_i müßte einen anderen Typ haben. Auch in dem Fall muß die Auswertung abgebrochen werden.

Die zweite Regel befaßt sich mit der speziellen Funktion APPLY. Sie kann nicht als eine primitive Funktion realisiert werden, denn die haben keinen Zugriff auf die Register des Interpreters. Den braucht APPLY aber, da die Funktion eine neue Aufruf–Fortsetzung erzeugt. So wird die Operation, die vom Interpreter implizit nach der Auswertung aller Elemente einer Kombination ausgeführt wird, für Scheme–Programme auch explizit nutzbar.

$$
\begin{array}{l}
\langle \textsc{primop}{:}\psi, \mathsf{call}{:}\langle\langle v_1, \ldots\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\qquad \rightarrow \quad \langle \psi(v_1, \ldots), \kappa, \rho, \sigma\rangle \\
\langle \textsc{apply}, \mathsf{call}{:}\langle\langle v_{op}, v_1, \ldots\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\qquad \rightarrow \quad \langle v_{op}, \mathsf{call}{:}\langle\langle v_1, \ldots\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\langle \textsc{callcc}, \mathsf{call}{:}\langle\langle v\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\qquad \rightarrow \quad \langle v, \mathsf{call}{:}\langle\langle \textsc{escape}{:}\langle \alpha, \kappa\rangle\rangle, \kappa\rangle, \rho, \sigma[\alpha \mapsto \textsc{unspecified}]\rangle \\
\qquad \qquad \text{mit einer bisher unbenutzten Adresse } \alpha \\
\langle \textsc{escape}{:}\langle \alpha, \kappa'\rangle, \mathsf{call}{:}\langle\langle v\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\qquad \rightarrow \quad \langle v, \kappa', \rho, \sigma\rangle \\
\langle \textsc{closure}{:}\langle \alpha, m, n, E, \langle \phi_1, \ldots, \phi_k\rangle\rangle, \mathsf{call}{:}\langle\langle v_1, \ldots, v_m\rangle, \kappa\rangle, \rho, \sigma\rangle \\
\qquad \rightarrow \quad \langle E, \kappa, \langle \phi_1, \ldots, \phi_k, \langle \beta_1, \ldots, \beta_{m+n}\rangle\rangle, \sigma'\rangle \\
\qquad \qquad \text{mit bisher unbenutzten Adressen } \beta_1, \ldots, \beta_{m+n} \text{ und} \\
\qquad \qquad \sigma' = \sigma[\begin{array}{ccc}
\beta_1 & \mapsto & v_1, \\
 & \vdots & \\
\beta_m & \mapsto & v_m, \\
\beta_{m+1} & \mapsto & \textsc{undefined}, \\
 & \vdots & \\
\beta_{m+n} & \mapsto & \textsc{undefined}
\end{array}]
\end{array}
$$

Abbildung 11: Fortsetzungsregeln für Funktionsaufrufe

$$\langle E, \kappa, \rho, \sigma[\beta \mapsto v, \ldots] \rangle \quad \to \quad \langle E, \kappa, \rho, \sigma \rangle$$

wenn $\{\beta, \ldots\}$ nicht leer ist
und $\beta, \ldots$ nicht in E, κ, ρ oder σ vorkommen

Abbildung 12: Garbage collection-Regel

Die CALLCC–Funktion benötigt ebenfalls eine eigene Regel, denn sie erzeugt nicht nur eine neue Fortsetzung, sondern liest vorher auch noch das Fortsetzungs–Register aus. Dessen Inhalt wird benötigt, um daraus eine Fortsetzungs–Funktion zu erzeugen.

Für die Ausführung dieser Funktionen ist die nächste Regel zuständig. Normalerweise würde κ in die neue Konfiguration übernommen. Fortsetzungs–Funktionen erlauben es aber, diesen normalen Kontrollfluß zu manipulieren. Sie ermöglichen es, zu einer vorher zwischengespeicherten Stelle im Programm zurückzukehren. Dazu enthalten sie eine andere Fortsetzung, hier κ'. Die wird statt κ in die neue Konfiguration eintragen.

Die letzte Regel ist auch die umfangreichste. Sie befaßt sich mit den in Scheme geschriebenen `lambda`–Funktionen. Diese Regel verändert jedes Element der Konfiguration. In das Programm–Register wird der übersetzte Rumpf E der Funktion eingetragen. Das Ausführungsumgebung–Register nimmt die um einen Rahmen ergänzte Ausführungsumgebung aus der Funktion auf. Dieser neue Rahmen enthält die Adressen der Argumente $v_1, \ldots, v_n$ aus der Aufruf–Fortsetzung und gegebenenfalls noch weitere für lokale Variablen. Das Fortsetzungs–Register erfährt seine normale Änderung. Im Speicher–Register wird schließlich eine neue Speicher–Funktion σ' eingetragen, welche die um neue Zellen für Argumente und lokale Variablen ergänzte alte Speicher–Funktion σ ist.

Nun ist auch nachvollziehbar, weshalb die Aufruf–Fortsetzung keine Ausführungsumgebung enthält: Alle außer einer der für ihre Auswertung zuständigen Regeln hinterlassen einen Literal im Programm–Register. Dieser führt zum Aufruf einer weiteren Fortsetzung, die dann, falls nötig, eine Ausführungsumgebung enthält. Die verbleibende Regel bezieht sich auf `lambda`–Funktionen, die selbst ihre benötigte Ausführungsumgebung mitbringen.

2.6.3 Garbage Collection

In den bisher vorgestellten Regeln wurde der Speicher nicht verändert, der Inhalt einer Zelle geändert oder neue Zellen hinzugefügt. Als Menge betrachtet ist σ dabei nur gewachsen. In einer realen Implementation ist das ein Problem: Die maximale Anzahl der benutzbaren Speicherzellen ist dort in der Regel begrenzt.

Doch viele der im Laufe einer Berechnung angeforderten Speicherzellen werden bald nicht mehr gebraucht. Sie sind zum Beispiel Teil von Datenstrukturen, die als Zwischenergebnisse erzeugt und nirgendwo sonst abgespeichert wurden. Oder auf sie wurde durch Adressen in Rahmen von Ausführungsumgebungen verwiesen, die nach Beendigung der entsprechenden Funktion nicht mehr benötigt werden.

Der Inhalt solcher Zellen kann den Lauf des Programmes nicht mehr beeinflussen. Daher ist es zulässig, diese Zellen und ihren Inhalt aus der Konfiguration zu entfernen. Dadurch wird die Menge σ wieder kleiner.

Diese Regel passt immer auf den Zustand des Interpreters. In einer Implementation wird sie vorwiegend dann zum Einsatz kommen, wenn der Speicher knapp wird.

3 Implementation

Hier wird die Implementation des Scheme–Systems in Java besprochen. Zunächst wird dazu die Sprache Java selbst betrachtet. Im Anschluß daran werden dann die Teile der Implementation selbst genauer betrachtet und erläutert.

Der vollständige Quellcode, die mit `javadoc` daraus erzeugte Dokumentation und ein `jar`–Archiv mit den `class`–Dateien befinden sich auf der CD, die dieser Arbeit beiliegt.

3.1 Was Java bietet

In diesem Abschnitt wird ein kurzer Überblick über die Implementations–Sprache Java gegeben. An einigen Stellen wird bereits kurz auf Unterschiede und Gemeinsamkeiten von Scheme und Java eingegangen.

3.1.1 Typsystem

Wie in Scheme so gibt es auch in Java atomare und zusammengesetzte Typen. Allerdings können Java–Programme neue sogenannte *Objekt–Typen* definieren.[5] Dabei kann für den neuen Typ festgelegt werden, dass seine Elemente eine Teilmenge der Elemente eines anderen bilden.

Jeder Objekt–Typ definiert eine Schnittstelle. Diese besteht aus den Funktionen, die auf Elemente dieses Typs angewandt werden können. Sie enthält deren Namen und ihre Ergebnis– und Parameter–Typen. Die Schnittstelle eines Typs bestimmt daher, wie das restliche System seine Elemente benutzen kann.

Objekt–Typen kommen in mehreren Variationen vor. Zunächst gibt es die *Interfaces*[6], die nur Konstanten und Deklarationen[7] von Funktionen enthalten. Dann gibt es die *Klassen* (engl. *Classes*), die auch eine Definition[8] der Funktionen angeben müssen. Nur Werte vom Klassen–Typ können erzeugt werden. Diese heißen dann *Instanzen* ihres Typs oder allgemein *Objekte*. Zwischen den Interfaces und Klassen stehen die abstrakten Klassen. Sie können wie Interfaces keine Instanzen haben und dürfen abstrakte Funktionen deklarieren, verhalten sich aber sonst wie Klassen.

Die erwähnte Teilmengen–Relation wird durch Vererbung hergestellt, von der ebenfalls mehrere Formen existieren. So können Klassen beliebig viele Interfaces *implementieren*. Ein Interface kann beliebig viele andere Interfaces und eine (Unter–)Klasse genau eine andere (Ober–)Klasse *erweitern*. Nur die Wurzel des so gebildeten Klassenbaumes hat keine Ober–Klasse. Diese Wurzel bildet die Klasse `java.lang.Object`. Sie ist Teil der zu Java gehörenden Bibliothek.

Besondere zusammengesetzte Typen sind die *Feld–Typen* (engl. *Arrays*). Sie sind zwar ebenfalls Objekt–Typen und erweitern die Klasse `Object`, können aber darüber hinaus nicht explizit an der Vererbung teilnehmen: Sie können keine Ober–Klassen selbst definierter Klassen sein. Allerdings respektieren sie bestehende Beziehungen ihrer Element–Typen: Der Typ eines Feldes von Elementen einer Unter–Klasse erweitert implizit den Typ eines Feldes von

[5] Genaugenommen müssen sie das sogar, da keine Funktionen außerhalb solcher Typen existieren können.

[6] Hier sind mir die Begriffe ausgegangen. Ich werde das deutsche Wort für das abstrakte Konzept und das englische für die konkrete Implementation in Java verwenden.

[7] Die *Deklaration* einer Funktion enthält nur die Informationen, die für die Schnittstelle benötigt werden. Sie bestimmt, wie eine Funktion aufzurufen ist.

[8] Die *Definition* einer Funktion enthält zusätzlich zu den Daten der Deklaration noch den Rumpf, definiert also das Verhalten.

Elementen der entsprechenden Ober–Klasse.

Da die atomaren Typen weder Klassen noch Interfaces sind, können sie folglich nicht erweitert oder im Sinne der Vererbung implementiert werden und sind auch nicht im Klassenbaum vertreten.

3.1.2 Variablen

In Java haben sowohl Variablen als auch ihre Werte einen Typ. Diese beiden werden als *statischer* und *dynamischer* Typ der Variablen bezeichnet.

Die atomaren Typen stehen in keiner Relation zueinander oder zu Objekt–Typen. Daher ist für Variablen dieser statischen Typen auch der dynamische Typ festgelegt. Folglich muß ihr Wert diese Information nicht mehr enthalten und kann daher effizienter dargestellt und manipuliert werden.

Variablen atomarer Typen haben Wert–Semantik. Eine Zuweisung an sie überschreibt ihren alten Wert und kopiert den zugewiesenen. Die Werte zweier solcher Variablen können zwar im Sinne des `==`–Operators äquivalent sein, sind aber verschiedene Instanzen ihres Typs.

Variablen eines Objekt–Typs dagegen sind Referenzen auf namenlose Objekte eines kompatiblen Typs. Eine Zuweisung läßt das Objekt selbst unangetastet, die Variable verweist danach einfach auf ein anderes. Mehrere Variablen können demnach auf das selbe Objekt verweisen. In dem Falle sind sie `==`–Äquivalent.

Die Referenz–Semantik erlaubt es, die Teilmengen–Relation der Objekt–Typen auszunutzen. So kann eine Variable, die den Typ einer Ober–Klasse trägt auch auf Objekte beliebiger Unter–Klassen verweisen. Eine Variable vom Interface–Typ kann auf Instanzen aller Klassen verweisen, die dieses Interface implementieren. Der dynamische Typ einer solchen Variablen ist also nicht festgelegt und kann sich sogar im Laufe der Zeit durch Zuweisungen ändern.

3.1.3 Funktionen

Im Gegensatz zu Scheme, wo Funktionen als freie Werte auftreten, sind sie in Java immer an einen Objekt–Typ gebunden. Das ist auch einer der Gründe dafür, Variablen einen statischen Typ zuzuordnen: Durch dessen Schnittstelle wird festgelegt, welche Funktionen für die Werte dieser Variablen zur Verfügung stehen. So besteht nicht – oder zumindest weit weniger – die Gefahr, eine Funktion mit Werten eines falschen Typs aufzurufen.

Funktionen operieren in der Regel[9] auf einer Instanz des Typs, zu dem sie gehören. Daher benötigen sie eine Referenz darauf, die als Parameter übergeben werden muß. Wegen der besonderen Rolle, die diese Instanz spielt, wird eine besondere Syntax beim Funktionsaufruf verwendet: Sie wird nicht in der Parameter–Liste eingetragen, sondern vor den Funktionsnamen gestellt: `i.f(x)`. Der erste Parameter ist hier das `i`, sein Wert ist in der Funktion als `this` verfügbar.

Der statische Typ einer Variablen kann allerdings ein Interface oder eine abstrakte Klasse sein. Dann enthält er zwar die Deklaration der Funktionen, aber möglicherweise keine Definition. Folglich ist der statische Typ nicht hinreichend, um das Verhalten einer Funktion festzulegen. Der dynamische Typ hingegen wird durch ein Objekt bestimmt. Als solches ist es Instanz einer Klasse, die zum statischen Typ kompatibel ist und alle Funktionen definieren muß. Funktionen in Java sind also *polymorph* im dynamischen Typ ihres impliziten Parameters.

Zusätzlich dazu erlaubt es Java, in einer Klasse oder einem Interface mehrere Funktionen gleichen Namens mit verschiedenen Parameter–Listen zu haben: Java–Funktionen sind zusätzlich polymorph in den statischen Typen ihrer expliziten Parameter.

Statische Polymorphie ist allerdings nur "syntaktischer Zucker", der es erlaubt, den selben Namen für verschiedene Funktionen mit ähnlichen Aufgaben zu verwenden. Ihr Einfluß auf das Programm verschwindet mit seiner Übersetzung. Dynamische Polymorphie dagegen hat auch bei der späteren Ausführung des Programmes noch Einfluß auf dessen Verhalten. Sie spielt bei der Implementation der Scheme–Regeln eine große Rolle.

3.1.4 Ausnahmen

Ausnahmen (engl. *Exceptions*) bieten eine Möglichkeit, mehrere geschachtelte Funktionsaufrufe zu verlassen. Das ist zum Beispiel nützlich, wenn in der innersten Funktion ein Fehler aufgetreten ist, der dort nicht behandelt werden kann. Dann wird eine Ausnahme signalisiert (*geworfen*), was den Abbruch der Funktionsausführung bewirkt. Dieser Vorgang setzt sich die Aufrufliste hinauf fort, bis sich eine Funktion bereiterklärt, die Ausnahme zu behandeln. Sie wird dort aufgefangen und kann Auskunft über die Ursache geben.

[9] Die Ausnahme der Regel bilden die statischen Funktionen. Sie können auch ohne eine Instanz aufgerufen werden und kennen demnach auch kein `this`.

3.1.5 Speicherverwaltung

Werte atomaren Typs sind in ihren Variablen enthalten. Der Code zur Verwaltung dieses Speichers wird vom Java–Compiler erzeugt. Objekte dagegen sind niemals selbst Inhalt einer Variablen. Auf sie wird nur verwiesen. Dennoch müssen auch sie irgendwo im Speicher stehen. Und der muß verwaltet werden.

Bei der Erzeugung eines Objekts wird Speicher mit `new` explizit angefordert. Um die spätere Freigabe muß sich das Programm nicht kümmern — kann es auch nicht, da Java das nicht vorsieht. Dennoch können auf einem System mit begrenztem Speicher Programme ausgeführt werden, deren Laufzeit nicht beschränkt ist. Dazu wird Speicher mit Objekten, auf die keine Referenzen mehr existieren, vom Laufzeit–System wieder freigegeben. Dieser Vorgang wird *Garbage–Collection* genannt.

Erfreulicherweise ist dies auch das Verhalten, das von einem Scheme–System gefordert wird. Die – durchaus nicht triviale – Aufgabe der Implementation eines Garbage–Collectors ist also durch die Wahl der Implementations–Sprache Java schon gelöst.

3.2 Implementation von Scheme

Objekte in Java müssen mit `new` erzeugt werden. Dabei ist natürlich der genaue Typ des angelegten Objekts bekannt. Das ist aber unter Umständen unerwünscht oder nicht möglich. Zum Beispiel, wenn der konkrete Typ ein Implementations–Detail sein soll und seine Instanzen vom Programm nur durch öffentliche Schnittstellen manipuliert werden. Weiterhin erzeugt `new` immer ein neues Objekt. Auch das ist manchmal nicht erwünscht, zum Beispiel für Singletons und Flyweights [13].

All diese Probleme mit der direkten Erzeugung kommen im hier vorgestellten Scheme–System vor: die abstrakte Klasse `Reference` versteckt zwei konkrete Klassen, die Interfaces `Pair` und `List` werden beide von der konkreten Klasse `PairOrList` implementiert, die aber als solche nie in Erscheinung tritt. Das einzige Objekt vom Typ `Empty` ist ein Singleton, `ScmBoolean` und `ScmChar` sind Flyweights.

Eine Lösung für diese Probleme ist die Verwendung von statischen `create`–Funktionen. Diese können die richtigen Konstruktoren aufrufen oder Referenzen auf vorgefertigte Objekte zurückgeben. Solche Funktionen können in abstrakten und konkreten Klassen als statische Memberfunktionen hinzugefügt werden. Das geht leider nicht in Interfaces wie `Pair` und `List`. In dem Falle muß auf eine zusätzliche Hilfsklasse, eine Factory zurückgegriffen werden, die eine oder mehrere statischen `create`–Funktionen beherbergen kann.

Um die Erzeugung von Objekten im Programm konsistent zu haben, wird hier weitgehend auf direkte Konstruktor–Aufrufe verzichtet. Auch Objekte, die direkt erzeugt werden könnten, werden in der Regel von `create`–Funktionen erstellt. Das erhöht zusätzlich die Unabhängigkeit von Schnittstelle und Implementation.

3.2.1 Zustand

Der Zustand des Scheme–Systems verteilt sich auf den Speicher, die Übersetzungs– und die Ausführungsumgebungen. Für den Speicher ist allerdings nichts zu implementieren, er wird vom Java–System zur Verfügung gestellt und verwaltet.

Bleiben also noch die Umgebungen.

Übersetzungsumgebungen. Die Übersetzungsumgebungen sind als verkettete Liste von Instanzen der Klasse

`MScheme.environment.StaticEnvironment`

implementiert. Da zur Verkettung Referenzen benutzt werden (müssen), signalisiert hier Javas `null` das Ende der Liste.

Diese Objekte stellen sich als assoziative Felder dar, die durch Symbole indiziert werden. Zur Implementation der Bindungsrahmen wird die Klasse `java.util.Hashtable` benutzt.

Schließlich gibt es noch zwei Zähler, deren Werte bei der Erzeugung neuer lexikalischer Adressen benutzt werden. Der erste gibt den Index des aktuellen Rahmens an, der zweite die Anzahl der bereits erzeugten lexikalischen Adressen. Beide Werte lassen sich zwar auch leicht berechnen, was aber – abhängig von der Größe der Umgebung – mehr oder weniger lange dauert. Hier wurde auf Kosten des Speicherbedarfs auf diese Berechnung verzichtet.

Die Klasse hat mehrere Konstruktoren. Der erste hat keine Parameter und erzeugt eine leere Umgebung mit nur einem Rahmen. Ein weiterer erwartet eine Referenz auf eine andere Umgebung und erzeugt eine neue innere.

Zwei weitere Konstruktoren erlauben es zusätzlich noch, in der neu erzeugten sofort einige Variablen anzulegen. Sie spielen bei der Übersetzung von `lambda`–Ausdrücken eine Rolle und signalisieren einen Fehler, wenn Symbole mehrfach vorkommen.

Natürlich lassen sich auch nach der Erzeugung noch neue Bindungen zu der Umgebung hinzufügen. Dazu bietet die Schnittstelle der Klasse einige Lese- und Schreibfunktionen, die allerdings jeweils lexikalische Adressen oder Syntax–Steuerzeichen liefern beziehungsweise erwarten.

Ausführungsumgebungen. Ausführungsumgebungen sind in der Klasse

`MScheme.environment.DynamicEnvironment`

implementiert. Sie bestehen in erster Linie aus einer Referenz auf ein Feld von Referenzen auf Bindungsrahmen. Jeder dieser Rahmen ist wiederum ein Feld von Referenzen auf Werte. Es werden daher zwei Indizes benötigt, um auf die Werte zuzugreifen. Die sind in den lexikalischen Adressen enthalten, die als Argument an Schreib– oder Lesefunktionen übergeben werden müssen.

Der Erzeugung dienen mehrere `create`–Funktionen. Eine statische und parameterlose erzeugt leere Ausführungsumgebungen mit jeweils einem Rahmen. Zwei weitere, nicht statische erlauben die Erzeugung von inneren Umgebungen vorgegebener Größe, eine der beiden trägt sogar sofort einige Werte ein.

Bei der Erzeugung von inneren Umgebungen wird ein neues, um ein Element größeres Feld von Referenzen auf Bindungsrahmen angelegt. Der Inhalt des entsprechenden Feldes der äußeren Umgebung wird hineinkopiert und das neue Element mit der Referenz auf einen neuen Bindungsrahmen in der als Argument übergebenen Größe initialisiert. Diese Größe ist bekannt; sie läßt sich aus der entsprechenden Übersetzungsumgebung ablesen.

Die Implementation der Ausführungsumgebungen ist der formalen Definition also treuer, als die der Übersetzungsumgebungen. Dort wurde das formale Tupel durch eine verkettete Liste implementiert, hier durch ein Feld. Das muß dann zwar kopiert werden, erlaubt aber schnelleren Zugriff über lexikalische Adressen. Diese Implementationstechnik wird in [3] als Display–Technik bezeichnet.

Für einen der Bindungsrahmen ist die Größe jedoch vorher unbekannt und kann sich sogar im Laufe der Zeit ändern: für den äußersten Rahmen, den der globalen Umgebung. Die Änderung der Größe eines Feldes ist in Java jedoch nicht möglich. Es muß neu erzeugt werden. Und dazu müßte der Speicher geändert werden.

Wäre der globale Rahmen also ebenfalls ein Feld, so könnte er seine Größe nicht ändern, da möglicherweise noch Referenzen auf den alten existieren.

Doch auch hier bietet die Klassenbibliothek wieder eine geeignete Lösung. Die Klasse `java.util.Vector` ist ein Container für Objekte, der wachsen kann. Die Implementation der Ausführungsumgebungen enthält daher zusätzlich zu dem oben beschriebenen Feld noch eine Referenz auf einen `Vektor`, der den globalen Rahmen darstellt.

3.2.2 Daten

Die Daten scheinen die passivste Komponente des Systems zu sein. Dennoch ist ihre Schnittstelle die umfangreichste. Das hängt damit zusammen, dass alle Funktionen des Scheme–Systems mit Daten arbeiten und daher vielfältige Anforderungen an diese gestellt werden. Insbesondere die unterschiedlichen Typsysteme von Java und Scheme machen eine Reihe von Funktionen notwendig, um den dynamischen Typ festzustellen oder als statischen bereitzustellen, um seine Schnittstelle nutzen zu können. Weitere Funktionen existieren für die drei Äquivalenz–Prädikate und die beiden Ausgabe–Methoden (`write` und `display`) von Scheme. Schließlich gibt es noch drei Funktionen, welche zum Daten→Programme–Übersetzer gehören.

Der Basistyp aller Werte ist das Interface

`MScheme.Value`.

Es definiert die Schnittstelle, die alle Werte erfüllen müssen. Dieses Interface ist die Implementation der formalen Menge **Wert**.

Die abstrakte Klasse

`MScheme.values.ValueDefaultImplementations`.

implementiert bereits nahezu die ganze Schnittstelle. Wenn im folgenden Text vom "Standard–Verhalten" oder der "Standard–Implementation" einer Funktion die Rede ist, so ist das in dieser Klasse implementierte Verhalten gemeint.

Diese Zweiteilung in ein Interface und eine abstrakte Klasse ist das Ergebnis verschiedener Versuche, Paare und Listen als getrennte Typen aber durch eine gemeinsame Klasse implementieren zu können. Die dafür notwendige Mehrfachvererbung erlaubt Java nur auf diesem Wege.[10]

[10] Darüber hinaus macht sie die Schnittstelle als Interface lesbar. Die `.java`–Datei der abstrakten Klasse enthält so viel Text, das die Schnittstelle darin völlig unter geht.

Wie in Abbildung 3 auf Seite 19 dargestellt, können Werte als Literale in Programmen auftreten. Aus diesem Grunde erweitert `ValueDefaultImplementations` die Klasse `Result` und implementiert die `getValue`–Funktion so, dass sie `this` zurück gibt.

Einige Hilfsfunktionen. Der Scheme–Report schreibt vor, dass Literale in Programmen konstant sein sollen. Die Konstruktor–Funktionen in Scheme und `read` liefern jedoch alle veränderbare zusammengesetzte Werte. Um Werte zu Konstanten zu machen, gibt es daher Funktion `getConst` in `Value`. Ihre Standard–Implementation gibt einfach `this` zurück, da atomare Werte automatisch unveränderbar sind und die meisten Scheme–Typen in diese Kategorie fallen.

In `if`–Ausdrücken wird das erste Argument als boole'scher Wert betrachtet. Die Umsetzung in den Java–Typ `boolean` geschieht dabei durch die Funktion `isTrue`. Deren Standard–Implementation gibt immer `true` zurück.

Diese Funktion ist eigentlich nur ein Vergleich mit dem Wert für FALSE. Es macht aber dennoch Sinn, sie in der Schnittstelle aller Werte zu haben: So kann jeder Wert auch ohne Kenntnis der beiden boole'schen Konstanten als `boolean` betrachtet werden.

Typecast Funktionen. Da Scheme weder Variablen noch Funktionesparametern einen Typ zuordnet, sind diese in Java als Referenzen auf das Interface `Value` implementiert. Sollen solche Objekte aber von primitiven Funktionen also Java–Programmen manipuliert werden, so muß deren dynamischer Typ über eine geeignete Referenz verfügbar sein.

Die Typcast–Operation, die Java zu diesem Zweck bietet, ist leider nicht das was hier benötigt wird. Sie wirft bei Typfehlern eine Ausnahme, die nur wenig Information zu ihrer Ursache enthält. Im interaktiven Scheme–System ist es aber wünschenswert, den erwarteten Typ und vorgefundenen Wert mitgeteilt zu bekommen.

Die Typen, die im Scheme–System vorkommen, sind von vornherein bekannt – der Scheme–Report schreibt sie vor. Daher kann das Interface `Value` Typcast–Funktionen für alle diese Typen bereitstellen. Das Standard–Verhalten dieser Funktionen ist es, einen Fehler ebenfalls durch Werfen einer Ausnahme zu signalisieren. Diese enthält dann aber eine Referenz auf den verursachenden Wert und eine Beschreibung des eigentlich erwarteten Typs.

Jede der verschiedenen konkreten Klassen, die die Scheme–Typen darstellen, muß die ihr entsprechende Funktion überschreiben und eine Referenz auf sich selbst zurück liefern.

So kann die Verwendung der Java–Typcast–Operation für Scheme–Daten nahezu völlig vermieden werden.

Neben diesen Typcast– gibt es auch noch Typabfrage–Funktionen in `Value`. Deren Standard–Implementation liefert immer `false`. Auch hier ist das Verhalten der entsprechenden Funktion in den konkreten Klassen zu ändern.

In den meisten Fällen ist klar, welche dieser Funktionen von konkreten Typen überschrieben werden müssen. Daher wird das in den folgenden Beschreibungen der einzelnen Klassen in der Regel nicht mehr explizit erwähnt.

Äquivalenz–Prädikate. Im Scheme–Report werden drei Äquivalenz–Prädikate definiert. `eq?` unterscheidet am stärksten, `eqv?` trennt etwas weniger scharf und `equal?` vergleicht Listen und Vektoren elementweise.

Für jedes dieser Scheme–Prädikate gibt es eine entsprechende Funktion in `Value`: `eq`, `eqv` und `equal`. Die Standard–Implementation von `eq` benutzt den `==`–Operator um die beiden Verweise zu vergleichen. Die beiden anderen Funktionen rufen jeweils die nächstfeinere auf. So muß eine konkrete Klasse nur eine dieser Funktionen neu implementieren, die gröberen verhalten sich dann automatisch korrekt.

intern→extern–Übersetzung. In Scheme gibt es zwei Ausgabe–Funktionen. Daher gibt es auch `write` und `display` in der Schnittstelle für Werte. Die Standard–Implementation von `display` delegiert ihre Aufgabe einfach an `write`, da die meisten Werte von beiden gleich ausgegeben werden. `write` dagegen hat kein sinnvolles Standard–Verhalten und ist daher als einzige Funktion nicht in der Basisklasse implementiert.

Beide erwarten als Argument einen `java.io.Writer`, in den die Ausgabe geschrieben wird.

Daten→Programm-Übersetzung. Da Werte, wenn sie implizit als Programme betrachtet werden nur Literale sind, muß die Übersetzung explizit geschehen. Dazu gibt es zwei Funktionen in der Schnittstelle.

Die beiden Funktionen `getCompiled` und `getSyntax` sind Bestandteile des rekursiven Übersetzers. Als solche benötigen sie Zugriff auf eine Übersetzungsumgebung. Eine Referenz auf eine solche wird ihnen daher als Argument übergeben.

`getCompiled` bildet die externe Schnittstelle zum Übersetzer. Ihre Standard–Implementation ignoriert die übergebene Übersetzungsumgebung und gibt `getConst()` zurück, da die meisten Scheme–Werte in Programmen zu Literalen werden.

`getSyntax` wird dagegen intern beim Übersetzen von Listen benutzt. Sie wird für deren Kopf aufgerufen und liefert ein `Syntax`–Objekt zurück, dessen `translate`–Funktion dann den Rest der Liste bearbeitet. Die Standard–Implementation geht davon aus, das die Liste ein einfacher Funktionsaufruf ist und erzeugt daher ein `ProcedureCall`–Objekt, das den Listenkopf zwischenspeichert.

Das Resultat eines Aufrufs von `getCompiled` hat zwar schon den richtigen Typ, ist aber noch nicht ganz fertig. Es kann noch verzögerte lexikalische Adressen enthalten. Diese werden erst bei einem zweiten Durchlauf durch das Programm ausgewertet. Darauf wird bei der Beschreibung der Programm–Klassen noch einmal näher eingegangen.

Wahr, Falsch und die leere Liste. Die beiden Klassen

`MScheme.values.ScmBoolean`

und

`MScheme.values.Empty`

haben insgesamt nur drei Instanzen: Die beiden boole'schen Konstanten für TRUE und FALSE und die leere Liste EMPTY.

Die Klasse `ScmBoolean` überschreibt `isTrue`, damit die Funktion für den Wert FALSE auch `false` zurück gibt.

`Empty` überschreibt `getCompiled` um einen Fehler zu signalisieren, denn die leere Liste ist kein korrekter Ausdruck.

Zeichen. Die Instanzen der Klasse

`MScheme.values.ScmChar`

enthalten jeweils ein Zeichen vom Java–Typ `char`.

Die Klasse überschreibt `eqv`, da der Scheme–Report die `eqv?`–Äquivalenz von Zeichen fordert. Das Prädikat `eq?` darf "gleiche" Zeichen unterscheiden. In der gegenwärtigen Implementation tut es das auch.

Zahlen. Der Scheme–Report definiert eine ganze Hierarchie numerischer Typen, von ganzen bis zu komplexen Zahlen. Allerdings müssen nicht alle diese Typen implementiert werden. Und da dieses Thema allein eine nicht zu vernachlässigende Komplexität besitzt, wird im vorliegenden System bisher nur die Menge der ganzen Zahlen dargestellt. Die Klasse

`MScheme.values.ScmNumber`

benutzt dazu die Bibliotheksklasse `java.math.BigInteger`.

Auch hier wird die Funktion `eqv` überschrieben. Für den Vergleich wird `BigInteger.equals` benutzt.

Symbole. Symbole, also Instanzen der Klasse

`MScheme.values.Symbol`

dienen in Scheme–Programmen als Bezeichner. Als Namen sind sie atomar und unveränderbar, obwohl sie aus einer Zeichenfolge bestehen. Java–Strings verhalten sich ähnlich. Daher werden sie als Symbole benutzt.

Sehr nützlich ist dabei die Funktion `String.intern()`. Sie erlaubt es, Zeichenketten zu vereinheitlichen. Dazu verwaltete die Klasse `String` eine Liste aller Zeichenketten, für die bereits einmal `intern` aufgerufen wurde. Wird die Funktion nun für eine Zeichenkette aufgerufen, die zeichenweise mit einer bereits bekannten identisch ist, so wird die aus der Liste zurückgegeben. Dadurch sind solche internalisierten Zeichenketten genau dann `Object.equals`–identisch, wenn die Verweise auf sie sie `==`–identisch sind.

Die von Scheme geforderte `eq?`–Identität von Symbolen kann hier also einfach durch die `==`–Identität der Verweise auf die Java–Zeichenketten implementiert werden. Nicht so einfach wäre es, die `Symbol`–Instanzen selbst identisch zu machen. Denn dazu müßte die Funktionalität von `String.intern()` für Symbole nachgebildet werden.

Aber das ist nicht notwendig. Alle drei Scheme–Äquivalenz–Prädikate sind als Funktionen im Interface `Value` enthalten und lassen sich überschreiben. Was in der Klasse `Symbol` dann auch geschieht. Diese Klasse ist allerdings die einzige, in der die `eq`–Funktion neu implementiert wird.

Da Symbole in Programmen eine besondere Rolle spielen, überschreibt diese Klasse sowohl `getCompiled` als auch `getSyntax`. In beiden Fällen wird die Übersetzung durch einen Zugriff auf die Übersetzungsumgebung realisiert. Die im Symbol enthaltene Zeichenkette dient dabei als Index.

Funktionen. Die abstrakte Klasse

`MScheme.values.Function`

erweitert die Schnittstelle der Basisklasse `ValueDefaultImplementations` um eine abstrakte `call`–Funktion. Sie ist die Basis einer großen Hierarchie verschiedener abstrakter und konkreter Klassen.

Die `call`–Funktion eines Objekts wird aufgerufen, wenn es im Scheme–System als Operator benutzt wurde. Sie erwartet als Argumente eine Referenz auf die Register des Interpreters und eine Scheme–Liste, in der die Scheme–Argumente enthalten sind. Ihr Ergebnis ist das `Code`–Objekt, das die Aktion des nächsten Berechnungsschritts bestimmt. Häufig ist dieses Objekt ein Literal.

Einige Helferklassen. Da jede Funktion nur passende Argument–Listen akzeptiert, gibt es die von `Function` abgeleitete Klasse

`MScheme.values.functions.CheckedFunction`.

Sie implementiert die `call`–Funktion der Basisklasse, definiert dafür aber zwei neue abstrakte. Eine ist `checkedCall`. Sie hat die selbe Signatur wie `call` und wird nach erfolgreicher Prüfung der Argument–Liste aufgerufen. Die zweite Funktion ist `getArity`. Sie liefert eine Referenz auf eine Instanz der Klasse

`MScheme.util.Arity`

zurück.

Dieses Objekt beschreibt die erlaubten Längen von Argument–Listen. Möglich sind in Scheme zum einen Listen mit fester Länge und zum anderen solche mit einer Minimal–Länge.

Weitere Helferklassen bilden die Basis für null– bis dreistellige Funktionen. Sie sind von `CheckedFunction` abgeleiteten aber Sie implementiert jeweils die `checkedCall`–Funktion der Basisklasse, die eine Liste als Argument erwartet. Dafür deklariert sie eine neue abstrakte Funktion, die statt dessen null bis drei Werte erwartet. Enthält die Scheme–Argumentenliste beim Aufruf die korrekte Anzahl Werte, so werden sie an die neue Funktion weitergereicht, ansonsten wird ein Fehler signalisiert. Diese Klassen sind die Basis von den meisten primitiven Scheme–Funktionen.

Primitive Funktionen. Diese Funktionen sind die Implementation der verschiedenen PRIMOP:ψ aus der formalen Definition.

Schon in Abschnitt 1.3.1 wurde erwähnt, dass die primitiven Funtionen einer Tabelle entnommen werden.

Doch wie soll diese Tabelle erzeugt werden? Es gibt 83 eingebaute Funktionen und vermutlich kommen noch mehr dazu.

Ein übliches Vorgehen ist es, ein großes `switch`–Statement zu verwenden, in dem alle diese Funktionen als `case` implementiert sind. Zusätzlich dazu wird eine Tabelle angelegt, in der unter dem Index, der im `case`–Label benutzt wird, der Name der entsprechenden Funktion steht.

Allerdings bringt das einige Nachteile mit sich. So sind zum Beispiel die Namen nur durch eine Zahl mit ihren Funktionen verbunden, und diese Zahl kommt in der Namenstabelle möglicherweise nicht einmal explizit vor. Ein versehentlich verursachter Fehler hätte fatale Folgen, da die Funktionen ihre Namen nicht kennen. Darüber hinaus müssen alle Funktionen mit den selben (Java–)Argumenten auskommen. Das könnte durch mehrere `switch`–Statements in verschiedenen Funktionen oder Klassen zwar ausgeglichen werden, was aber wiederum die Verwaltung der Indizes erheblich erschweren würde.

Eine völlig andere Möglichkeit bietet Javas Reflection–API. Sie erlaubt es, die Schnittstelle einer Klasse aus dem laufenden Java–System heraus zu untersuchen. Werden die primitiven Funktionen nun zum Beispiel als statische Funktionen in einer Klasse implementiert, so kann man auf diesem Wege auf deren Namen, Rückgabe– und Parameter–Typen zugreifen und sie sogar aufrufen. Dieser Name ist aufs engste mit der Funktion verbunden: es ist der Name, unter dem sie im Java–System bekannt ist. Werden nun die Zeichen, die in Scheme–Symbolen erlaubt, in Java–Bezeichnern aber verboten sind, geeignet umschrieben, so kann dieser Name auch in das Scheme–System übernommen werden.

Dennoch hat auch dieser Ansatz seine Nachteile. Es wird hier Arbeit zur Laufzeit des Programms erledigt, die bereits bei der Übersetzung der Funktions–Sammelklasse geleistet werden könnte.

Daher wurde hier eine Methode benutzt, die die Vorteile der beiden eben vorgestellten Vorgehensweisen vereinigt. Es wurde ein Java–Programm implementiert, das die Reflection–API benutzt, um die `switch`–Statements und die Tabellen automatisch zu erzeugen. Dieses Programm ist in der Klasse `ParseBuiltins` implementiert. Es untersucht die Schnittstelle von `Builtins` und erzeugt daraus die Datei `BuiltinTable.java`.

Fortsetzungsfunktionen. Die Instanzen der Klasse

`MScheme.machine.ContinuationFunction`

erlauben es, Fortsetzungen wie einstellige Funktionen zu behandeln. Dazu enthalten sie eine Referenz auf eine Fortsetzung, also eine Instanz von `Continuation`.

Wird eine solche Funktion aufgerufen, so macht sie die gespeicherte Fortsetzung wieder zur aktuellen und reicht ihr Argument daran weiter.

Die in ihrer formalen Form ESCAPE:$\langle\alpha, \kappa\rangle$ vorkommende Adresse α ist in der Implementation implizit gegeben.

Benutzerdefinierte Funktionen. Auch die Klasse

`MScheme.code.CompiledLambda.Closure`

ist von `CheckedFunction` abgeleitet. `Closure` ist eine lokale Klasse. In Java kennt die Instanz einer solchen Klasse die Instanz der umschließenden Klasse, in der sie erzeugt wurde. Das ist hier sehr nützlich, da ein `Closure`–Objekt alle Informationen "seines" `CompiledLambda`–Objekts benötigt. Alles was es noch selbst explizit speichern muß, ist die bei seiner Erzeugung aktuelle Ausführungsumgebung.

Beim Aufruf einer solchen Funktion wird zu der enthaltenen Ausführungsumgebung ein neuer Rahmen hinzugefügt, in den die Argumente eingetragen werden. Diese neue Ausführungsumgebung wird zur aktuellen gemacht und der übersetzte Rumpf zur Auswertung zurückgegeben.

Schemes IO-System. Zur Kommunikation mit seiner Umgebung benutzt Scheme Ports. Deren Implementation besteht aus der abstrakten Klasse

`MScheme.values.Port`,

die die Funktion `isPort` überschreibt, und den beiden konkreten Klassen

`MScheme.values.InputPort`

und

`MScheme.values.OutputPort`.

Für die IO-Operationen in Java ist darin jeweils ein `Reader` beziehungsweise `Writer` zuständig.

Der `InputPort` enthält einen klassischen Lexer/Parser, der eine Zeichenkette einliest und einen Parsebaum erzeugt. In diesem Falle ist das bereits die interne Darstellung der Daten. Dieser Parser wird von Schemes `read`–Funktion benutzt.

Er wurde hier durch einen von Hand implementierten, rekursiv absteigenden LL(1)–Parser realisiert. Im Laufe der Entwicklung wurden auch Parser–Generatoren wie JFlex [21] oder ANTLR getestet. Eine bereits vorliegende Implementation in Java, die Einfachheit der zugrundeliegenden Grammatik und die Tatsache, das ein Leser sich mit einer weiteren Sprache auseinander zu setzen hätte, führten schließlich zur Vermeidung externer Werkzeuge.

Wird der dagegen ein `OutputPort` von Schemes `write`– oder `display`–Funktion aufgefordert, Daten auszugeben, so reicht er seinen `Writer` an eine der bereits besprochenen Ausgabe–Funktionen aus der Schnittstelle der Werte weiter. Deren Implementationen erledigen dann die eigentliche Ausgabe.

Die Asymmetrie der Implementation folgt aus der Tatsache, dass bei der Ausgabe die Typinformation bereits vorliegt, die beim Einlesen erst aus einer Zeichenkette zurückgewonnen werden muß.

Zusammengesetzte Daten. Der Scheme–Report schreibt in Abschnitt 3.4 vor, dass Literale nicht verändert werden dürfen. Das ist automatisch erfüllt, wenn die Werte atomar sind. Zusammengesetzte Datenstrukturen dagegen könnten mit Hilfe der Mutations–Funktionen verändert werden. Um in solch einer Situation gegebenenfalls einen Fehler melden zu können, muß die entsprechende Information vorliegen. Auch sollte im Sinne der Modularität *eine* Funktion den Fehler melden, um das Verhalten einfacher austauschbar zu machen. Aus diesem Grunde wurde die abstrakte Klasse

`MScheme.values.Compound`

implementiert. Ihr wird im Konstruktor mitgeteilt, ob der Wert konstant ist. Bevor dann eine Mutations-Operation ausgeführt wird, muß die von dieser Klasse bereitgestellte `modify`–Funktion aufgerufen werden. Ist der Wert konstant, wird ein entsprechender Fehler signalisiert.

Für alle Klassen des mit `Compound` beginnenden Astes der Vererbungshierarchie ist die Standard–Implementation von `getConst` nicht mehr korrekt. Daher wird diese Funktion in `Compound` überschrieben. Für konstante Werte ist sie nach wie vor die Identität, für nicht konstante wird `getConstCopy` aufgerufen. Diese abstrakte Funktion muß von allen Klassen, die `Compound` erweitern, geeignet implementiert werden.

Vektoren. Die Vektoren in Scheme haben einiges mit den Feldern in Java gemeinsam. Ihre Größe wird bei der Erzeugung festgelegt und kann danach nicht mehr verändert werden. Ihre Elemente können dagegen modifiziert werden. Dem entsprechend ist in der Klasse

```
MScheme.values.ScmVector
```

auch ein solches Feld enthalten. Es nimmt Verweise auf beliebige Scheme–Werte auf.

Weitere Funktionen dieser Klasse bilden die Basis für die Scheme–Funktionen `list->vector` und `vector->list`. Die `equal`–Funktion ist überschrieben, da Vektoren vom Schemes `equal?` komponentenweise verglichen werden. Natürlich sind auch die von der abstrakten Basisklasse `Compound` geforderten Funktionen implementiert.

Wie schon `Empty`, so überschreibt auch diese Klasse die `getCompiled`–Funktion um einen Fehler zu melden. Auch Vektoren sind keine korrekten Ausdrücke.

Zeichenketten. Scheme–Zeichenketten sind Vektoren mit eingeschränktem Element–Typ. Im Gegensatz zu Javas Zeichenketten, den `String`s können Schemes verändert werden. Die Zeichenketten–Klasse

```
MScheme.values.ScmString
```

benutzt deshalb wie die Vektoren ein Feld. Das nimmt allerdings atomare Java–Zeichen auf.

Paare und Listen. Sowohl `List` als auch `Pair` sind Interfaces. Beide erweitern `Value` und haben aber sonst nicht viel gemeinsam:

Paare haben zwei Elemente. Die können gelesen und, wenn das Paar nicht konstant ist, auch verändert werden.

Listen sind in Scheme eine Abstraktion und in Java daher nur ein Interface. Sie bestehen aus einer Kette von (null oder mehr) Paaren, die in der leeren Liste endet. Sie haben eine Länge und, falls sie nicht leer sind, einen Kopf und einen Rest, wobei der Rest immer auch eine Liste ist. Nur nicht–leere

Listen lassen sich in Programme übersetzen. Ihre Schnittstelle enthält keine Mutations-Operationen.

Dennoch werden beide Interfaces von einer konkreten Klasse implementiert:

`MScheme.values.PairOrList`.

Diese Klasse erweitert `Compound`, da ein Paar eine zusammengesetzte Datenstruktur ist. Das Interface `List` wird zusätzlich noch von der Klasse `Empty` implementiert.

Die Implementation von `isPair` und `toPair` ist einfach, denn jede Instanz von `PairOrList` ist ein Paar. Auch `toList` ist leicht zu implementieren, wenn `isList` gegeben ist.

Im Gegensatz zu `isPair` macht `isList` aber eine Aussage über die Datenstruktur, deren Teil das Paar ist. Daher ist für sie eine komplexere Implementation notwendig.

Ein Wert ist eine Liste, wenn er die leere Liste oder ein Paar mit einer Liste als zweitem Element ist. Diese rekursive Definition beschreibt eine naive Implementation der `isList`–Funktion: In Paaren liefert sie einfach das Ergebnis eines rekursiven Aufrufs für deren zweites Element zurück. In `Empty` ist diese Funktion ebenfalls überschrieben und liefert `true`, für alle anderen Typen liefert sie `false`.

Aber: In Scheme können Paare modifiziert werden. Dadurch können zyklische Datenstrukturen entstehen, zum Beispiel, indem das zweite Element eines Paares so verändert wird, dass es auf das Paar selbst verweist. Solche Strukturen sind dann keine Listen, da nirgendwo die leere Liste vorkommt. Die eben beschriebene Implementation von `isList` könnte das aber nicht in endlicher Zeit feststellen. Sie gerät dann je nach Implementation entweder in eine Endlosschleife oder erreicht irgendwann die maximale Rekursionstiefe des Java–Systems und bricht mit einem Fehler ab.

Eine bessere Lösung fand sich in [26] als (rekursives) Scheme–Programm. Eine (iterative) Java–Version ist in `PairOrList.isList()` implementiert. Der Trick besteht darin, zwei Verweise durch die Paar–Kette laufen zu lassen, einen doppelt so schnell wie den anderen. In Listen trifft der schnellere bald auf die leere Liste, die das Ende markiert und der Algorithmus ist mit einer positiven Antwort beendet.

Ist die Kette zyklisch mit beliebig langem nicht–periodischen Präfix, so haben dennoch irgendwann beide Verweise den Präfix hinter sich gelassen. Danach kreisen sie gemeinsam im Zyklus. Nach jedem Schritt des langsameren werden die beiden verglichen und nach höchstens einem kompletten Umlauf verweisen sie auf das selbe Paar. Das beendet den Algorithmus mit einer negativen Antwort, es liegt keine Liste vor.

Eine dritte Möglichkeit, die Überprüfung zu beenden gibt es auch noch: Im Laufe der Berechnung kann es zu einem Typfehler kommen, wenn das zweite Element eines Paares weder die leere Liste noch ein weiteres Paar ist. Auch dann wird eine negative Antwort gegeben.

`toList` wird nur dann erfolgreich beendet, wenn die Antwort von `isList` positiv ausfiel. Da nur auf diesem Wege Verweise auf das Interface `List` erzeugt werden, können nun andere Funktionen, die auf Listen operieren ohne weitere Fehlerkontrollen und somit effizienter implementiert werden.

Übersetzung von Listen. Die Übersetzung von Listen in Scheme ist recht kompliziert. Das liegt daran, dass ihr erstes Element und nicht ihr erstes Paar bestimmt, als was sie zu betrachten sind. Die in `PairOrList` implementierte Version von `getCompiled` ruft daher zuerst `getSyntax` für den Kopf der Liste auf. Diese Funktion gibt eine Referenz auf ein Objekt zurück, das das Interface

`MScheme.Syntax`

implementiert. Die einzige Funktion darin ist `translate`. Deren Aufgabe ist es, den Rest der Liste und eine Übersetzungsumgebung entgegenzunehmen und daraus ein `Code`–Objekt zu erzeugen.

Die eigentliche Übersetzung wird also von den verschiedenen Klassen erledigt, die `Syntax` implementieren. Da jede dieser Klassen nur Instanzen einer Programm–Klasse erzeugt, werden sie zusammen mit diesen im folgenden Abschnitt besprochen.

3.2.3 Programme

Alle Programm–Objekte implementieren das Interface

`MScheme.Code.`

Es enthält zwei Funktionen. Die erste, `executionStep`, wird vom Interpreter benutzt, um einen Schritt in der Auswertung des Programmes auszuführen. Die zweite Funktion, `force`, bildet den zweiten Teil des Daten→Programme–Übersetzers. Ihre Aufgabe ist es, die verzögerten lexikalischen Adressen in echte umzuwandeln.

Diese Klasse hat einen direkten Bezug zu der formalen Spezifikation des letzten Kapitels: Sie entspricht der Menge **Programm**, die in Abbildung 3 auf Seite 19 definiert wird. Die im Folgenden vorgestellten Klassen korrespondieren mit den darin vorkommenden Präfixen. Die verschiedenen Implementationen von `executionStep` führen schließlich jeweils eine oder mehrere der Reduktionen aus Abbildung 9 auf Seite 31 aus.

Im Folgenden werden bereits die verschiedenen konkreten Fortsetzungsklassen vorgestellt. Deren gemeinsame, abstrakte Basisklasse `Continuation` wird später noch ausführlich besprochen. Hier nur soviel vorweg: Das Erzeugen einer Fortsetzung mit `new` oder einer `create`–Funktion legt diese bereits auf dem Fortsetzungs–Stack ab. Wird sie später mit einem Wert aufgerufen, so wird sie zuerst vom Fortsetzungs–Stack entfernt. Anschließend wird die in der Basisklasse abstrakte Funktion `executionStep` ausgeführt.

Literale und andere Werte. Wie bereits zu Beginn von Abschnitt 3.2.2 angemerkt, können Daten als Literale Teil von Programmen sein. Dazu erweitert `ValueDefaultImplementations` die Klasse

`MScheme.machine.Result`.

Diese Klasse spielt eine zentrale Rolle im Scheme–System: Sie ruft in der Ausführungsschleife des Interpreters die Fortsetzungen auf. Der Wert, der an die Fortsetzung übergeben wird, ist das Ergebnis eines Aufrufs von `getValue`. Diese in `Result` abstrakte Funktion wird in konkreten Klassen geeignet implementiert. In `ValueDefaultImplementations` zum Beispiel gibt sie `this` zurück.

Zu den Literalen ist die Syntax–Klasse

`MScheme.syntax.Quote`.

assoziiert. Ihre `translate`–Funktion macht mit dem ersten Element der Argument–Liste das, was die Standard–Implementation von `getCompiled` tut. Damit können auch solche Werte in Programme aufgenommen werden, deren `getCompiled`–Funktion überschrieben wurde.

Da Literale immer Blätter in der baumförmigen Datenstruktur des Programmes sind, endet bei ihnen der rekursive Abstieg von `force`. Da sie auch keinen weiteren Zustand enthalten, der ge`force`d werden müßte, ist diese Funktion in `ValueDefaultImplementations` als Identität implementiert, das heißt, sie gibt `this` zurück.

Lexikalische Adressen. Die Variablen im Scheme–Programm werden bei der Übersetzung zu Objekten der abstrakten Klasse

`MScheme.environment.Reference.`

Auch sie erweitert `Result`. Dabei wird die `getValue`–Funktion als Zugriff auf die Ausführungsumgebung implementiert.

Für die Zusammenarbeit mit dem Übersetzer gibt es zwei konkrete Implementationen dieser Klasse:

`MScheme.environment.DelayedReference`

speichert das Symbol und die Übersetzungsumgebung, in der es später existieren muß. Instanzen dieser Klasse haben aber in vollständig übersetzten Programmen nichts mehr zu suchen. Dort kommen nur noch Instanzen der Klasse

`MScheme.environment.ForcedReference`

vor, die lexikalische Adressen sind. Sie bestehen aus den beiden Indizes für Rahmen und Eintrag.

Kombinationen. Die Klasse

`MScheme.code.Application`

enthält ein Feld mit Referenzen auf weitere `Code`–Objekte.

Die Implementation der `force`–Funktion aktualisiert jedes Element des Feldes mit dem Ergebnis von dessen `force`–Aufruf. Anschließend liefert sie `this` zurück.

Instanzen dieser Klasse werden von Objekten vom Typ

`MScheme.syntax.ProcedureCall`

erzeugt. Diese Klasse unterscheidet sich von allen anderen konkreten, von `Syntax` abgeleiteten Klassen: Sie ist kein Singleton.

In Kombinationen ist der Kopf der Liste keine Sonderform, die bei der Übersetzung verschwindet, sondern der aufzurufende Operator. Dem entsprechend speichert das `ProcedureCall`–Objekt diesen Wert zwischen. In der `translate`–Funktion werden dann sowohl der Operator als auch die Operanden übersetzt. Die so erzeugte Sequenz von `Code`–Objekten wird in dem oben bereits erwähnten Feld einer Instanz von `Application` abgelegt. Die Flexibilität einer aus Paaren aufgebauten Liste wird hier nicht mehr benötigt – im Gegenteil: Es ist sinnvoll in konstanter Zeit auf jedes Element zugreifen zu können und ihre Anzahl zu kennen.

In der formalen Spezifikation sind mit dem Aufruf von Funktionen zwei Fortsetzungen assoziiert. So ist es auch in der Implementation. Die Auswertung geschieht ebenfalls von hinten nach vorn.

Liegt ein Aufruf mit Argumenten vor, so wird eine Instanz einer namenlosen Erweiterung von `Continuation` erzeugt. Als Objekt einer lokalen Klasse kennt sie das bei ihrer Erzeugung aktuelle `Application`–Objekt und damit die auszuwertende Teilprogramm–Sequenz. Darüber hinaus speichert die Fortsetzung noch den Index des nächsten auszuwertenden Arguments und eine (Scheme–)Liste mit den bereits ausgewerteten. Die Reihenfolge der Auswertung erlaubt es dabei, diese Liste in der richtigen Reihenfolge wachsen zu lassen.

Wird der Index schließlich 0 – bei Aufrufen ohne Argumente ist er das bereits zu Anfang – so wird eine

`MScheme.code.ApplyContinuation`

erzeugt. Sie speichert nur noch die Liste der ausgewerteten Argumente zwischen und wartet auf die Auswertung des Operators. Wird dieser schließlich an ihre `executionStep`–Funktion übergeben, so wird er mit `toFunction` als `Function` interpretiert und deren `call`–Funktion mit der gespeicherten Argumente–Liste aufgerufen.

Die in der formalen Spezifikation notwendigen vier Regeln für Kombinationen, Schiebe– und Aufruf–Fortsetzungen konnten in der Implementation zu einer Funktion verschmolzen werden: `Application.prepareNext`. Sowohl `Application.executionStep` als auch die `executionStep`–Funktion der namenlosen Fortsetzung benutzen diese Funktion.

`Lambda`–Ausdrücke. Ein übersetzter `lambda`–Ausdruck, also eine Instanz von

`MScheme.code.CompiledLambda`,

enthält eine Referenz auf den übersetzten Rumpf. Zusätzlich dazu ist auch die "Stelligkeit", also die Anzahl der erlaubten Parameter in Form einer Referenz auf ein `Arity`–Objekt in der Instanz abgespeichert. Als drittes Attribut enthält die Klasse noch die Größe der später beim Aufruf der erzeugten Funktion anzulegenden Ausführungsumgebung.

Erzeugt werden diese Objekte von der Syntax–Klasse

`MScheme.syntax.Lambda`.

Deren `translate`–Funktion überprüft die Scheme–Parameter–Liste und erzeugt ein entsprechendes `Arity`–Objekt. Danach wird die aktuelle Übersetzungsumgebung um einen Rahmen erweitert, der Bindungen der formalen Parameter an lexikalische Adressen enthält.

Die neue Übersetzungsumgebung, das `Arity`–Objekt und der bisher unbehandelte Rumpf des `Lambda`–Ausdrucks werden dann an eine der `create`–Funktionen von `CompiledLambda` übergeben. Dort wird der Rumpf mit Hilfe eines `Begin`–Syntax–Objekts übersetzt. Schließlich wird die später benötigte Größe der Ausführungsumgebung aus der Übersetzungsumgebung ausgelesen und in die entsprechende Membervariable eingetragen.

Da `CompiledLambda` wie schon die Literale und lexikalischen Adressen immer einen Wert erzeugt, ist die Klasse ebenfalls von `Result` abgeleitet. Die daher notwendige Implementation von `getValue` erzeugt ein neues `Closure`–Objekt.

`If`–Ausdrücke. Eine Instanz der Klasse

`MScheme.code.Selection`

speichert drei Referenzen auf weitere Teilprogramme. Die erste zeigt auf den Test–Ausdruck. Sein Wert entscheidet, ob das eine oder das andere der beiden restlichen Teilprogamme auszuwerten ist.

Erzeugt werden `Selection`–Objekte von der Syntax–Klasse

`MScheme.syntax.If`.

Dabei wird gegebenenfalls das optionale dritte Argument ergänzt. Die drei Argumente werden übersetzt und in ein neues `Selection`–Objekt eingetragen.

Da die Auswertung eines `if`–Ausdrucks immer zwei Schritte benötigt, ist auch hier eine Fortsetzung notwendig. Wie schon die Schiebe–Fortsetzung in

Kombinationen, so ist auch diese als lokale, namenlose Klasse in `Selection` implementiert.

Diese Fortsetzung wird von der Implementation von `executionStep` erzeugt, bevor sie den Test–Ausdruck zurück gibt. Nach dessen Auswertung überprüft die Fortsetzung in `executionStep` den erhaltenen Wert mit `isTrue` und stellt anschließend das entsprechende Teilprogramm zur weiteren Auswertung bereit.

`Set!`– und `define`–Ausdrücke. Eine Zuweisung wird in ein Objekt der Klasse

`MScheme.code.Assignment`

übersetzt. Es besteht aus der lexikalischen Adresse, an die zugewiesen wird und dem Programm, das den zuzuweisenden Wert berechnet.

Ein `Assignment`–Objekt wird von zwei Syntax–Klassen erzeugt. Die erste ist

`MScheme.syntax.Set`.

Die in dieser Klasse implementierte `translate`–Funktion geht davon aus, das die zu setzende Variable bereits existiert.

Aber auch

`MScheme.syntax.Define`.

erzeugt ein `Assignment`–Objekt. Die zusätzliche Aufgabe von `define` – das Erzeugen neuer Variablen – spielt nur bei der Übersetzung eine Rolle und wird daher vollständig in der `translate`–Funktion erledigt.

Auch hier hat die `executionStep`–Funktion nicht viel zu tun. Sie erzeugt eine neue Fortsetzung – wiederum Instanz einer lokalen, namenlosen Klasse – und gibt das Programm zurück, dessen Wert zugewiesen werden soll. Ist dessen Berechnung abgeschlossen, nimmt die `executionStep`–Funktion der Fortsetzung ihn entgegen und trägt ihn an der durch die lexikalische Adresse bezeichneten Stelle in der Ausführungsumgebung ein.

`Begin`–Ausdrücke. Die Instanzen der Klasse

`MScheme.code.Sequence`

unterscheiden sich in ihrer Struktur nicht von Kombinationen, wohl aber in ihrer Erzeugung und ihrem Verhalten.

Wie Kombinationen enthalten sie eine Sequenz von Teilprogrammen. Allerdings ist das erste Element der ursprünglichen Liste nicht Teil der Sequenz, denn die Instanz von

`MScheme.syntax.Begin`,

deren `translate`–Funktion `Sequence`–Objekte erzeugt, ist ein Singleton.

Sequenzen mit nur einem Element sind in Scheme erlaubt. Allerdings wäre es unsinnig, für sie ein `Sequence`–Objekt anzulegen; das einzige Teilprogramm kann ebenso gut für sich selbst stehen. Diese einfache Optimierung wird daher in der `create`–Funktion von `Sequence` vorgenommen.

Bei der Auswertung von Sequenzen kommt ebenfalls wieder eine lokale, namenlose Fortsetzungs–Klasse zum Einsatz. Der Ablauf beim Aufruf der Funktion `executionStep` ist analog zu dem in `Application`, auch hier gibt es wieder eine `prepareNext`–Funktion. Allerdings wird der Index hier hoch gezählt und vor der Auswertung des letzten Elements der Sequenz wird keine neue Fortsetzung erzeugt.

3.2.4 Interpreter

Nachdem nun die Programme vorgestellt sind, geht es nun um ihre Ausführung. Die vielen Regeln der formalen Spezifikation sind als dynamisch polymorphe Funktionen in den konkreten Programm– und Fortsetzungs–Klassen realisiert. Nun müssen sie nur noch aufgerufen werden.

Das geschieht in der Funktion `Machine.execute`. Dabei kommen Instanzen zweiter Helferklassen zum Einsatz. Zum einen die bereits kurz angesprochene Basisklasse für Fortsetzungen, zum anderen eine Klasse, die einige Register enthält.

Register. Die Instanzen der Klasse

`MScheme.machine.Registers`

enthalten zwei der vier Elemente des Zustandes $\langle E, \kappa, \rho, \sigma \rangle$ des Interpreters: Die aktuelle Fortsetzung κ und die Ausführungsumgebung ρ. Der Speicher σ wird vom Java–System verwaltet, ist also gar nicht greifbar. Das jeweils aktuelle Programm E ist in einer lokalen Variable der `Machine.execute`–Funktion enthalten.

Ein Verweis auf ein Objekt dieses Typs wird bei allen `executionStep`– und `Function.call`–Aufrufen weitergereicht. So haben zum Beispiel Zuweisungen und Variablen Zugriff auf die Ausführungsumgebung oder es können neue Fortsetzungen angelegt werden.

Fortsetzungen. Ein `Registers`–Objekt enthält genau den Teil des Zustands des Interpreters, der von Fortsetzungen gespeichert und bei ihrem Aufruf wieder zurückgeschrieben werden muß.

Um zu vermeiden, dass bei der Erzeugung einer neuen Fortsetzung immer zusätzlich noch ein neues `Registers`–Objekt erzeugt wird, erweitert die Klasse

`MScheme.machine.Continuation`

die Implementation von `Registers`.

Dem Konstruktor einer Fortsetzung muß eine Referenz auf ein `Registers`–Objekt übergeben werden. Dessen Inhalt wird kopiert und anschließend die gerade erzeugte als aktuelle Fortsetzung darin eingetragen.

Der Aufruf einer Fortsetzung geschieht von `Result.executionStep` aus über ihre `invoke`–Funktion. Darin werden zuerst die gespeicherten Register zurückgeschrieben und danach die von den verschiedenen konkreten Klassen entsprechend implementierte `executionStep`–Funktion aufgerufen. Diese unterscheidet sich von der gleichnamigen Funktion aus `Code` durch einen zusätzlichen Parameter vom Typ `Value`.

Trampolin. Die hier verwandte Art der Implementation wird von S. Ganz, et. al. in [14] als *Trampolined Style* bezeichnet:

> A trampolined program is organized as a single loop in which computations are scheduled and their execution allowed to proceed in discrete steps. [...]

Im Einzelnen passiert dabei folgendes:

Wird `Machine.execute` ein übersetztes Programm zur Ausführung übergeben, so wird die Referenz darauf zunächst in die entsprechende lokale Variable

– den “Instruction pointer” `current` – eingetragen. Danach wird ein neues `Registers`–Objekt erzeugt. Die initiale Ausführungsumgebung–Referenz wird einer Membervariablen der Klasse `Machine` entnommen. Als erste und zunächst einzige Fortsetzung wird eine Instanz der Klasse

`MScheme.machine.StopContinuation`

erzeugt. Sie erlaubt es, abzufragen ob sie bereits aufgerufen wurde.

So vorbereitet wird die Auswertung gestartet. Sie ist als `while`–Schleife implementiert, die beendet wird, wenn die `StopContinuation` aufgerufen wurde.

Innerhalb der Schleife wird `current.executionStep` aufgerufen und das Resultat wieder an `current` zugewiesen. Dabei können allerdings Ausnahmen geworfen werden – zum Beispiel, wenn versucht wird, auf den `car` einer Zahl zuzugreifen. Da sich ein interaktives Scheme–System in so einer Situation nicht einfach beenden sollte, werden solche Ausnahmen aufgefangen. Hat das ausgeführte Scheme–Programm eine Fehler–Behandlungs–Funktion eingetragen, so werden die in der Ausnahme enthaltenen Informationen daran weitergeleitet und die Auswertungsschleife wird fortgesetzt. Dabei wird die eingetragene Funktion gelöscht. So wird verhindert, dass ein Fehler in der Fehler–Behandlungs–Funktion eine Endlosschleife auslöst. Ein Fehler führt dann dazu, dass die aufgefangene Ausnahme wieder geworfen wird ohne behandelt worden zu sein.

3.2.5 Maschine

Die Klasse

`MScheme.machine.Machine`,

ist die Schnittstelle zwischen dem Java– und dem Scheme–System. Sie enthält die globale Umgebung als Instanz der Klasse

`MScheme.environment.Environment`,

die wiederum Verweise auf die globale Ausführungs– und Übersetzungsumgebung enthält.

Die Schnittstelle der Maschine enthält Funktionen um auf die verschiedenen Übersetzer und den Interpreter des Scheme–Systems zuzugreifen. Letzterer ist zwar größtenteils in den verschiedenen konkreten Programm–Klassen implementiert, aber die Maschine – beziehungsweise deren `execute`–Funktion – ist für die Ablaufsteuerung bei der Auswertung zuständig.

Im Konstruktor der Klasse werden einige für die jeweilige Maschine spezifische Funktionen und Werte an Scheme–Variablen gebunden. Danach wird der Inhalt der Variablen `Init.bootstrap` übersetzt und ausgeführt. Im Anschluß daran steht die Instanz zur Auswertung weiterer Ausdrücke zur Verfügung.

Falls die so erzeugte Maschine allerdings nicht von einem Java–Programm aus benutzt, sondern gleich interaktiv werden soll, reicht es, die `run`–Funktion aufzurufen. Darin wird der Inhalt der Variablen `Init.rep` ausgewertet. Diese enthält eine Scheme–Implementation einer Read–Eval–Print–Schleife.

4 Ergänzung und Ausblick

Bereits in der Einführung wurde auf die Definition der Sprache Scheme im Scheme–Report [19] verwiesen. Neben diesem eher akademischen Standard gibt es auch einen offiziellen: den *IEEE Standard for the Scheme Programming Language* [18] von 1991.

Implementation. Hilfreich bei der Entwicklung des Designs war Gamma, *et al.* [13]. Bei der Verbesserung und Änderung war Fowler [12] sehr nützlich.

Mit Analyse und Übersetzung von Programmen im Allgemeinen beschäftigen sich Aho, *et al.* [2, 3] sowohl theoretisch als auch praktisch.

Kelsey [20] untersucht verschiedene Implementationstechniken für Sprachen, in denen endrekursive Aufrufe beliebiger Tiefe möglich sind. Feeley und Miller [10] beschreiben eine für Scheme und ähnliche Sprachen entworfene virtuelle Maschine. Morrisett, *et al.* [25] behandeln die Speicherverwaltung in Systemen mit Garbage Collection.

Zur Implementationssprache Java gibt es eine Reihe von Büchern, die sich mit der Sprache selbst, der virtuellen Maschine und den zugehörigen Bibliotheken beschäftigen, zB [4, 9, 15, 23]. Sehr nützlich ist natürlich auch die Online–Dokumentation. Diese ist – neben vielen weiteren Informationen – im Internet unter `http://java.sun.com/` zu finden.

Kontrollfluß–Operationen. Die Sprache Scheme selbst bietet mit `if`, `begin` und dem Funktionsaufruf nicht viele Möglichkeiten, den Kontrollfluß im Programm zu beeinflussen. Da aber Fortsetzungen als Funktionen und damit als Werte manipuliert werden können, erlaubt es die Sprache, eigene Kontrollfluß–Operationen zu definieren. Einige Beispiele und weitere Betrachtungen dazu sind in Queinnec [27] und Sabry, *et al.* [29] zu finden.

Eine Implementation von `dynamic-wind` ist mit Klasse

```
MScheme.machine.WindContinuation
```

bereits in MScheme [30] enthalten.

Eine interressante Alternative zu Fortsetzungen sind die in Hieb, *et al.* [16] beschriebenen Subcontinuations. Eine experimentelle Implementation von `spawn` ist ebenfalls bereits [30] enthalten.

Multithreading. Die Möglichkeit, Multithreading mit Hilfe von Fortsetzungen zu realisieren wird bereits in Wand [31] erörtert. In Kumar, *et al.* [22] werden Subcontinuations unter diesem Aspekt betrachtet. Auch der hier verwandte *Trampolined Style* kann als Basis für Multithreading dienen. Das wird in Ganz, *et al.* [14] beschrieben. Eine weitere Möglichkeit stellen *Futures* dar. Sie werden in Flanagan, *et al.* [11] und Moreau [24] dargestellt.

Makros. Schemes Makros wurden in dieser Arbeit bisher nicht behandelt. Nur eingebaute Sonderformen wurden besprochen. Hilsdale und Friedmann [17] machen deutlich, warum: Der im Scheme–Report beschriebene Makro–Mechanismus ist so komplex, dass er sogar Turing–vollständig ist.

Eine Implementation, zum Beispiel basierend auf Clinger [7, 6] oder Rees [28], steht daher noch aus. Zur Zeit ist nur eine experimentelle Implementation eines einfachen, nicht-hygienischen Macrosystems realisiert. Sie erlaubt es, die Daten vor dem eingebauten Überzetzer durch beliebige Scheme–Funktionen bearbeiten zu lassen.

Eine Erweiterung des Makro–Konzepts beschreibt Bawden [5]. Nach den Funktionen werden dort auch Makros zu 'first-class'-Werten.

A Symbolverzeichnis

$$
\begin{array}{rcl}
E & \in & \mathbf{Programm} \\
v & \in & \mathbf{Wert} \subset \mathbf{Programm} \\
f & \in & \mathbf{Funktion} \subset \mathbf{Wert} \\
c & \in & \mathbf{Zeichen} \\
z & \in & \mathbb{Z} \\
I & \in & \mathbf{Symbol} \\
\psi & \in & \mathbf{Wert}^* \to \mathbf{Wert} \\
\\
\kappa & \in & \mathbf{Fortsetzung} \\
\\
P & \in & \mathbf{\ddot{U}U} \\
\Phi & \in & \mathbf{\ddot{U}R} \\
B & \in & \mathbf{Bindung} \\
A & \in & \mathbf{LexAdresse} \subset \mathbf{Programm} \\
S & \in & \mathbf{Syntax} \\
\\
\rho & \in & \mathbf{AU} \\
\phi & \in & \mathbf{AR} \\
\alpha & \in & \mathbf{Adresse} \\
\sigma & \in & \mathbf{Adresse} \to \mathbf{Wert} \\
\\
i, j, n, m & \in & \mathbb{N}
\end{array}
$$

Literatur

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Struktur und Interpretation von Computerprogrammen*. Springer, 4th edition, 2001.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau – Teil 1*. Addison-Wesley, 1988.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau – Teil 2*. Addison-Wesley, 1988.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, May 2000.

[5] Alan Bawden. First-class macros have types. In *Conf. Rec. POPL '00: 27th ACM Symp. Princ. of Prog. Langs.*, pages 133–141, Boston, Massachusetts, USA, January 2000. ACM, ACM Press.

[6] William Clinger. Macros in Scheme. *Lisp Pointers*, IV(4):25–28, October 1991.

[7] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, Orlando, FL USA, January 1991.

[8] William D. Clinger. Proper tail recursion and space efficiency. *ACM SIGPLAN Notices*, 33(5):174–185, May 1998.

[9] Gary Cornell and Cay S. Horstmann. *Core Java*. Prentice Hall, 1996.

[10] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, Nice, France, June 1990.

[11] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Conf. Rec. POPL '95: 22th ACM Symp. Princ. of Prog. Langs.*, San Francisco, CA USA, January 1995. ACM, ACM Press.

[12] Martin Fowler. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, September 2000.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Professional Computing Series. Addison-Wesley, 1995.

[14] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. *International Conference on Functional Programming*, pages 18–27, 1999.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, July 2000.

[16] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *LISP and Symbolic Computation*, 7(1):83–110, January 1994.

[17] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation–passing style. *Proceedings of the Workshop on Scheme and Functional Programming 2000*, pages 53–60, September 2000.

[18] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*. IEEE standard 1178-1990, 1991.

[19] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[20] Richard A. Kelsey. Tail-recursive stack disciplines for an interpreter. Technical report, NEC Research Institute, March 1993.

[21] Gerwin Klein. *JFlex User's Manual*, August 1999. Version 1.2.2.

[22] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *LISP and Symbolic Computation*, 10(3):223–236, May 1998.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.

[24] Luc Moreau. The semantics of Scheme with future. *ICFP'96 5/96 PA, USA. 1996 ACM*, May 1996.

[25] Greg Morriset, Matthias Felleisen, and Robert Harper. Abstract models of memory management. *Proc. 1995 ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, 1995.

[26] Bryan O'Sullivan. Re: pair? and list?, November 1995. posted as a news article in 'comp.lang.scheme'.

[27] Christian Queinnec. A library of high level control operators. *Lisp Pointers*, 6(4):11–26, October 1993.

[28] Jonathan Rees. The scheme of things: Implementing lexically scoped macros. *Lisp Pointers*, VI(1), January-March 1993.

[29] Amr Sabry and Matthias Felleisen. Reasoning with continuations III: A complete calculus of control, 1992. `http://www.cs.indiana.edu/hyplan/sabry/papers/reasoning-popl-sub.ps`.

[30] Marvin Sielenkemper. The MScheme homepage, October 2001. `http://mscheme.sourceforge.net/`.

[31] Mitchell Wand. Continuation–based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, August 1980.