

17 Die SECD-Maschine

Der λ -Kalkül ist als theoretisches Modell für berechenbare Funktionen lange vor der Erfindung des Computers entwickelt worden. Die Reduktionsregeln dienen dabei der Entwicklung von Beweisen über die Äquivalenz von λ -Termen. Damit der λ -Kalkül auch als Modell für die tatsächliche Ausführung von Programmen, auf dem Computer geeignet ist, fehlen noch zwei Zutaten: die direkte Definition von „eingebauten“ Werten und Operationen wie Zahlen und booleschen Werten sowie ein formales Auswertungsmodell. Dieses Kapitel stellt zunächst den *angewandten λ -Kalkül* vor, der den normalen λ -Kalkül um primitive Werte und Operationen erweitert, und dann die *SECD-Maschine*, ein klassisches Auswertungsmodell für die Call-by-Value-Reduktion. Angenehmerweise läßt sich die SECD-Maschine auch als Scheme-Programm implementieren, was ebenfalls in diesem Kapitel geschieht. Die SECD-Maschine kennt keine Zuweisungen; es folgt darum noch die Darstellung der *SECDH-Maschine*, die auch einen *Speicher* kennt und damit Zuweisungen korrekt modelliert.

17.1 Der angewandte λ -Kalkül

Abschnitt 16.3 zeigte bereits, daß sich auch boolesche Werte und Zahlen im λ -Kalkül durch λ -Terme darstellen lassen. Das ist zwar aus theoretischer Sicht gut zu wissen, auf Dauer aber etwas mühsam: Darum ist es sinnvoll, mit einer erweiterten Version des λ -Kalküls zu arbeiten, die solche „primitiven“ Werte direkt kennt. Abschnitt 16.3 hat gezeigt, daß eine solche Erweiterung nur syntaktischer Zucker ist, also die Ausdruckskraft des Kalküls nicht wirklich erhöht. Alle Erkenntnisse aus dem normalen λ -Kalkül bleiben also erhalten.

Ein solcher erweiterter λ -Kalkül heißt auch *angewandter λ -Kalkül*. Seine primitiven Werte und Operationen werden durch eine Σ -Algebra definiert:

Definition 17.1 (Sprache des angewandten λ -Kalküls $\mathcal{L}_{\lambda\mathcal{A}}$) Sei V eine abzählbare Menge von Variablen. Sei Σ ein Operationsalphabet und $(B, \{F_B \mid F \in \Sigma\})$ eine Σ -Algebra von *Basiswerten*. Die F_B heißen *Primitive*. Die Sprache des angewandten λ -Kalküls, die Menge der *angewandten λ -Terme*, $\mathcal{L}_{\lambda\mathcal{A}}$, ist die kleinste Menge mit folgenden Eigenschaften:

1. $V \subseteq \mathcal{L}_{\lambda\mathcal{A}}$
2. Für $e_0, e_1 \in \mathcal{L}_{\lambda\mathcal{A}}$ ist auch $(e_0 e_1) \in \mathcal{L}_{\lambda\mathcal{A}}$.
3. Für $x \in V, e \in \mathcal{L}_{\lambda\mathcal{A}}$ ist auch $(\lambda x.e) \in \mathcal{L}_{\lambda\mathcal{A}}$.
4. $B \subseteq \mathcal{L}_{\lambda\mathcal{A}}$

5. Für $e_1, \dots, e_n \in \mathcal{L}_{\lambda A}$ und $F \in \Sigma^{(n)}$ ist $(F e_1 \dots e_n) \in \mathcal{L}_{\lambda A}$.

Dabei heißen Terme der Form $(F e_1 \dots e_n)$ auch *primitive Applikationen*.

In diesem Kapitel dienen normalerweise die Zahlen als Basiswerte mit den üblichen Operationen wie $+$, $-$, $*$, $/$ etc. Damit sind Terme wie zum Beispiel $(+ (- 5 3) 17)$ möglich.

Im angewandten λ -Kalkül kommen zu den Werten aus Definition 16.12 die Basiswerte dazu:

Definition 17.2 (Werte im angewandten λ -Kalkül) Im angewandten λ -Kalkül heißen die Abstraktionen und Basiswerte kollektiv *Werte*. Ein λ -Term, der kein Wert ist, heißt *Nichtwert*.

Damit die primitiven Operationen auch tatsächlich eine Bedeutung bekommen, muß eine spezielle Reduktionsregel für sie eingeführt werden:

Definition 17.3 (δ -Reduktion)

$$(F e_1 \dots e_n) \rightarrow_{\delta} F_B(e_1, \dots, e_n) \quad e_1, \dots, e_n \in B$$

Diese Regel besagt, daß eine primitive Applikation, wenn alle Operanden Werte sind, durch Anwendung der entsprechenden Operation in der Σ -Algebra B reduziert werden kann. Damit wird z.B. der obige Beispielterm folgendermaßen reduziert:

$$(+ (- 5 3) 17) \rightarrow_{\delta} (+ 2 17) \rightarrow_{\delta} 19$$

17.2 Die einfache SECD-Maschine

Wie schon in Abschnitt 16.5 erwähnt, ist der Call-by-Value- λ -Kalkül ein Modell für die Auswertung von Scheme und viele andere Programmiersprachen. Allerdings ist Definition 16.15 strenggenommen etwas vage: Es wird immer nur der Subterm reduziert, der „möglichst weit links innen steht“, aber was das heißt, ist nicht genau definiert. Außerdem ist Reduktion zwar ein mächtiges formales Modell, entspricht aber nicht der Ausführungsmethode tatsächlicher Scheme-Implementierungen auf echten Prozessoren. Ein präzises und echten Maschinen deutlich näheres Modell ist die *SECD-Maschine*, erfunden schon in den 60er Jahren von Peter Landin [LANDIN 1964], und seitdem die Grundlage für zahllose Implementierungen von Call-by-Value-Sprachen. (Die Darstellung hier ist gegenüber Landins ursprünglicher Formulierung etwas modernisiert.)

Damit ein Programm aus dem angewandten λ -Kalkül mit der SECD-Maschine ausgewertet werden kann, muß es erst einmal in einen speziellen *Maschinencode* übersetzt oder „compiliert“ werden. Der Maschinencode besteht, anders als der λ -Kalkül, nicht aus geschachtelten Termen, sondern aus einer Folge von *Instruktionen*.

Definition 17.4 (Maschinencode) In der folgenden Definition ist I die Menge der Instruktionen und C ein Maschinencode-Programm:

$$\begin{aligned}
 I &= B \\
 &\quad \cup V \\
 &\quad \cup \{\text{ap}\} \\
 &\quad \cup \{\text{prim}_F \mid F \in \Sigma\} \\
 &\quad \cup V \times C \\
 C &= I^*
 \end{aligned}$$

Die Vereinigung von I ist dabei so zu verstehen, daß die vereinigten Mengen alle paarweise disjunkt sind, also z.B. $B \cap V = \emptyset$.

Ein Term aus dem angewandten λ -Kalkül wird mit Hilfe folgender Funktion in Maschinencode übersetzt:

$$\begin{aligned}
 \llbracket _ \rrbracket &: \mathcal{L}_{\lambda A} \rightarrow C \\
 \llbracket e \rrbracket &\stackrel{\text{def}}{=} \begin{cases} b & \text{falls } e = b \in B \\ v & \text{falls } e = v \in V \\ \llbracket e_0 \rrbracket \llbracket e_1 \rrbracket \text{ap} & \text{falls } e = (e_0 e_1) \\ \llbracket e_1 \rrbracket \dots \llbracket e_n \rrbracket \text{prim}_F & \text{falls } e = (F e_1 \dots e_n) \\ (v, \llbracket e_0 \rrbracket) & \text{falls } e = \lambda v. e_0 \end{cases}
 \end{aligned}$$

Die Übersetzungsfunktion „linearisiert“ einen λ -Term. Zum Beispiel bedeutet die Übersetzung $\llbracket e_0 \rrbracket \llbracket e_1 \rrbracket \text{ap}$ für einen Term $(e_0 e_1)$, daß zuerst e_0 ausgewertet wird, danach wird e_1 ausgewertet, und schließlich wird die eigentliche Applikation ausgeführt: Entsprechend steht ap für „Applikation ausführen“ und prim_F für „Primitiv F ausführen“. Basiswerte und Variablen werden im Maschinencode belassen. Ein λ -Term wird übersetzt in ein Tupel aus seiner Variable und dem Maschinencode für seinen Rumpf.

Durch die Linearisierung sind die Instruktionen schon in einer Liste in der Reihenfolge ihrer Ausführung aufgereiht. Insbesondere hat die Linearisierung den Begriff „links innen“ formalisiert: der jeweils am weitesten links innen stehende Redex steht in der Liste der Instruktionen vorn.

Beispiel:

$$\begin{aligned}
\llbracket \lambda f. \lambda x. \lambda y. f (+ x (* y 2)) \rrbracket &= (f, \llbracket \lambda x. \lambda y. f (+ x (* y 2)) \rrbracket) \\
&= (f, (x, \llbracket \lambda y. f (+ x (* y 2)) \rrbracket)) \\
&= (f, (x, (y, \llbracket f (+ x (* y 2)) \rrbracket))) \\
&= (f, (x, (y, \llbracket f \rrbracket \llbracket (+ x (* y 2)) \rrbracket \text{ap}))) \\
&= (f, (x, (y, f \llbracket (+ x (* y 2)) \rrbracket \text{ap}))) \\
&= (f, (x, (y, f \llbracket x \rrbracket \llbracket (* y 2) \rrbracket \text{prim}_+ \text{ap}))) \\
&= (f, (x, (y, f x \llbracket (* y 2) \rrbracket \text{prim}_+ \text{ap}))) \\
&= (f, (x, (y, f x \llbracket y \rrbracket \llbracket 2 \rrbracket \text{prim}_* \text{prim}_+ \text{ap}))) \\
&= (f, (x, (y, f x y \llbracket 2 \rrbracket \text{prim}_* \text{prim}_+ \text{ap}))) \\
&= (f, (x, (y, f x y 2 \text{prim}_* \text{prim}_+ \text{ap})))
\end{aligned}$$

Das Beispiel zeigt deutlich, wie der Rumpf der innersten Abstraktion in eine lineare Folge von Instruktionen übersetzt wird, die genau der Call-by-Value-Reduktionsstrategie entspricht: erst f auswerten, dann x , dann y , dann das Primitiv $*$ anwenden, dann $+$, und schließlich die Applikation durchführen.

Nun zur eigentlichen SECD-Maschine – sie funktioniert ähnlich wie ein Reduktionskalkül, operiert aber auf sogenannten *Maschinenzuständen*: die Maschine überführt also einen Maschinenzustand durch einen Auswertungsschritt in einen neuen Maschinenzustand. Ein Maschinenzustand ist dabei ein 4-Tupel aus der Menge $S \times E \times C \times D$ (daher der Name der Maschine). Die Buchstaben sind deshalb so gewählt, weil S der sogenannte *Stack*, E die sogenannte *Umgebung* bzw. auf englisch das *Environment*, C der schon bekannte Maschinencode bzw. *Code* und D der sogenannte *Dump* ist. Die formalen Definitionen dieser Mengen sind wie folgt; dabei ist W die Menge der Werte:

$$\begin{aligned}
S &= W^* \\
E &= \mathcal{P}(V \times W) \\
D &= (S \times E \times C)^* \\
W &= B \cup (V \times C \times E)
\end{aligned}$$

Der Stack ist dabei eine Folge von Werten. In der Maschine sind dies die Werte der zuletzt ausgewerteten Terme, wobei der zuletzt ausgewertete Term vorn bzw. „oben“ steht. Die Umgebung ist eine partielle Abbildung von Variablen auf Werte: sie ersetzt die Substitution in der Reduktionsrelation des λ -Kalküls. Anstatt daß Werte für Variablen eingesetzt werden, merkt sich die Umgebung einfach, an welche Werte die Variablen gebunden sind. Erst wenn der Wert einer Variablen benötigt wird, holt ihn die Maschine aus der Umgebung. Der Dump schließlich ist eine Liste früherer Zustände der Maschine: er entspricht dem Kontext im Substitutionsmodell.

Die Menge W schließlich entspricht dem Wertebegriff aus Definition 17.2: Die Basiswerte gehören dazu, außerdem Tripel aus $(V \times C \times E)$. Ein solches Tripel, genannt *Closure* – repräsentiert den Wert einer Abstraktion – es besteht aus der Variable einer Abstraktion, dem Maschinencode ihres Rumpfs und der Umgebung, die notwendig ist, um die Abstrakti-

on anzuwenden: Die Umgebung wird benötigt, damit die freien Variablen der Abstraktion entsprechend der lexikalischen Bindung ausgewertet werden können. Dies ist anders als im Substitutionsmodell, wo Variablen bei der Applikation direkt ersetzt werden und damit verschwinden. Eine Closure ist also einfach die Repräsentation einer Funktion.

Im Verlauf der Auswertung werden Umgebungen häufig um neue Bindungen von einer Variable an einen Wert erweitert. Dazu ist die Notation $e[v \mapsto w]$ nützlich. $e[v \mapsto w]$ konstruiert aus einer Umgebung e eine neue Umgebung, in der die Variable v an den Wert w gebunden ist. Hier ist die Definition:

$$e[v \mapsto w] \stackrel{\text{def}}{=} (e \setminus \{(v, w') \mid (v, w') \in e\}) \cup \{(v, w)\}$$

Es wird also zunächst eine eventuell vorhandene alte Bindung entfernt und dann eine neue hinzugefügt.

Um einen λ -Term e in die SECD-Maschine zu „injizieren“, wird er in einen Anfangszustand $(\varepsilon, \emptyset, \llbracket e \rrbracket, \varepsilon)$ übersetzt. Dann wird dieser Zustand wiederholt in die Zustandsübergangsrelation \hookrightarrow gefüttert. In der folgenden Definition von \hookrightarrow sind Bezeichner mit einem Unterstrich versehen, wenn es sich um Folgen handelt, also z.B. \underline{s} für einen Stack:

$$\hookrightarrow \in \mathcal{P}((S \times E \times C \times D) \times (S \times E \times C \times D))$$

$$(\underline{s}, e, b\underline{c}, \underline{d}) \hookrightarrow (b\underline{s}, e, \underline{c}, \underline{d}) \quad (17.1)$$

$$(\underline{s}, e, v\underline{c}, \underline{d}) \hookrightarrow (e(v)\underline{s}, e, \underline{c}, \underline{d}) \quad (17.2)$$

$$(b_n \dots b_1 \underline{s}, e, \text{prim}_F \underline{c}, \underline{d}) \hookrightarrow (b\underline{s}, e, \underline{c}, \underline{d}) \quad (17.3)$$

wobei $F \in \Sigma^{(n)}$ und $F_B(b_1, \dots, b_n) = b$

$$(\underline{s}, e, (v, \underline{c}')\underline{c}, \underline{d}) \hookrightarrow ((v, \underline{c}', e)\underline{s}, e, \underline{c}, \underline{d}) \quad (17.4)$$

$$(w(v, \underline{c}', e')\underline{s}, e, \text{ap } \underline{c}, \underline{d}) \hookrightarrow (\varepsilon, e'[v \mapsto w], \underline{c}', (\underline{s}, e, \underline{c})\underline{d}) \quad (17.5)$$

$$(w, e, \varepsilon, (\underline{s}', e', \underline{c}')\underline{d}) \hookrightarrow (w\underline{s}', e', \underline{c}', \underline{d}) \quad (17.6)$$

Die Regeln definieren eine Fallunterscheidung nach der ersten Instruktion der Code-Komponente des Zustands, bzw. greift die letzte Regel, wenn der Code leer ist. Der Reihe nach arbeiten die Regeln wie folgt:

- Regel 17.1 (die *Literalregel*) schiebt einen Basiswert direkt auf den Stack.
- Regel 17.2 (die *Variablenregel*) ermittelt den Wert einer Variable aus der Umgebung und schiebt diesen auf den Stack.
- Regel 17.3 ist die *Primitivregel*. Bei einer primitiven Applikation müssen so viele Basiswerte oben auf dem Stack liegen wie die Stelligkeit des Primitivs. Dann ermittelt die Primitivregel das Ergebnis der primitiven Applikation und schiebt es oben auf den Stack.
- Regel 17.4 ist die *Abstraktionsregel*: Das Tupel (v, \underline{c}') ist bei der Übersetzung aus einer Abstraktion entstanden. Die Regel ergänzt v und \underline{c}' mit e zu einer Closure, die auf den Stack geschoben wird.
- Regel 17.5 ist die *Applikationsregel*: Bei einer Applikation müssen oben auf dem Stack ein Wert sowie eine Closure liegen. (Zur Erinnerung: Eine Applikation kann nur ausgewertet werden, wenn eine Abstraktion vorliegt. Abstraktionen werden zu Closures

ausgewertet.) In einem solchen Fall „sichert“ die Applikation den aktuellen Zustand auf den Dump, und die Auswertung fährt mit einem leeren Stack, der Umgebung aus der Closure – erweitert um eine Bindung für die Variable – und dem Code aus der Closure fort.

- Regel 17.6 ist die *Rückkehrregel*: Sie ist anwendbar, wenn das Ende des Codes erreicht ist. Das heißt, daß gerade die Auswertung einer Applikation fertig ist. Auf dem Dump liegt aber noch ein gesicherter Zustand, der jetzt „zurückgeholt“ wird.

Hier ein Beispiel für den Ablauf der SECD-Maschine für den Term $((\lambda x.\lambda y.(+ x y)) 1) 2$:

$(\epsilon,$	$\emptyset,$	$(x, (y, x y \text{ prim}_+))$	$1 \text{ ap } 2 \text{ ap},$	$\epsilon)$
$\hookrightarrow ((x, (y, x y \text{ prim}_+), \emptyset),$	$\emptyset,$	$1 \text{ ap } 2 \text{ ap},$	$\epsilon)$	
$\hookrightarrow (1 (x, (y, x y \text{ prim}_+), \emptyset),$	$\emptyset,$	$\text{ap } 2 \text{ ap},$	$\epsilon)$	
$\hookrightarrow (\epsilon,$	$\{(x, 1)\},$	$(y, x y \text{ prim}_+),$	$(\epsilon, \emptyset, 2 \text{ ap}))$	
$\hookrightarrow ((y, x y \text{ prim}_+, \{(x, 1)\}),$	$\{(x, 1)\},$	$\epsilon,$	$(\epsilon, \emptyset, 2 \text{ ap}))$	
$\hookrightarrow ((y, x y \text{ prim}_+, \{(x, 1)\}),$	$\emptyset,$	$2 \text{ ap},$	$\epsilon)$	
$\hookrightarrow (2 (y, x y \text{ prim}_+, \{(x, 1)\}),$	$\emptyset,$	$\text{ap},$	$\epsilon)$	
$\hookrightarrow (\epsilon,$	$\{(x, 1), (y, 2)\},$	$x y \text{ prim}_+,$	$(\epsilon, \emptyset, \epsilon))$	
$\hookrightarrow (1,$	$\{(x, 1), (y, 2)\},$	$y \text{ prim}_+,$	$(\epsilon, \emptyset, \epsilon))$	
$\hookrightarrow (2 \ 1,$	$\{(x, 1), (y, 2)\},$	$\text{prim}_+,$	$(\epsilon, \emptyset, \epsilon))$	
$\hookrightarrow (3,$	$\{(x, 1), (y, 2)\},$	$\epsilon,$	$(\epsilon, \emptyset, \epsilon))$	
$\hookrightarrow (3,$	$\emptyset,$	$\epsilon,$	$\epsilon)$	

Die Zustandsübergangsrelation \hookrightarrow ist nun die Grundlage für die *Auswertungsfunktion* der SECD-Maschine, die für einen λ -Term dessen Bedeutung ausrechnet. Dies ist scheinbar ganz einfach:

$$\begin{aligned} \text{eval}_{SECD} &: \mathcal{L}_{\lambda A} \rightarrow B \\ \text{eval}_{SECD}(e) &= x \text{ wenn } (\epsilon, \emptyset, \llbracket e \rrbracket, \epsilon) \hookrightarrow^* (x, e, \epsilon, \epsilon) \end{aligned}$$

Diese Definition hat jedoch zwei Haken:

- Die Auswertung von λ -Termen terminiert nicht immer (wie zum Beispiel für den „Endlos-Term“ $(\lambda x.(x x)) (\lambda x.(x x))$), es kommt also nicht immer dazu, daß die Zustandsübergangsrelation bei einem Zustand der Form $(\epsilon, \emptyset, \llbracket e \rrbracket, \epsilon)$ terminiert.
- Das x aus dieser Definition ist nicht immer ein Basiswert – es kann auch eine Closure sein.

Der erste Haken sorgt dafür, daß die Auswertungsfunktion nur eine Relation im Sinne einer „partiellen Funktion“ ist. Meist wird trotzdem von einer Auswertungsfunktion gesprochen. Beim zweiten Haken, wenn x eine Closure ist, läßt sich mit dem Resultat nicht viel anfangen: Um die genaue Bedeutung der Closure herauszubekommen, müßte sie angewendet werden – das Programm ist aber schon fertig gelaufen. Es ist also gar nicht sinnvoll, zwischen verschiedenen Closures zu unterscheiden. Darum wird für die Zwecke der Auswertungsfunktion eine Menge Z der *Antworten* definiert, die einen designierten Spezialwert für Closures enthält:

$$Z = B \cup \{\text{function}\}$$

Damit läßt sich die Evaluationsfunktion wie folgt definieren:

$$eval_{SECD} \in \mathcal{L}_{\lambda A} \times \mathcal{Z}$$

$$eval_{SECD}(e) = \begin{cases} b & \text{falls } (\varepsilon, \emptyset, \llbracket e \rrbracket, \varepsilon) \hookrightarrow^* (b, e, \varepsilon, \varepsilon) \\ \text{function} & \text{falls } (\varepsilon, \emptyset, \llbracket e \rrbracket, \varepsilon) \hookrightarrow^* ((v, \underline{c}, e'), e, \varepsilon, \varepsilon) \end{cases}$$

17.3 Quote und Symbole

Dieses Kapitel wird ab hier Gebrauch von einer weiteren Sprachebene in DrScheme machen, nämlich Die Macht der Abstraktion - fortgeschritten. Diese Ebene muß mit dem DrScheme-Menü Sprache unter Sprache auswählen aktiviert sein, damit die Programme dieses Kapitels funktionieren.

Die entscheidende Änderung gegenüber den früheren Sprachebenen ist die Art, mit der die REPL Werte ausdrückt. (Diese neue Schreibweise, ermöglicht, die Programme des Interpreters, die als Werte repräsentiert sind, korrekt auszudrucken.) Bei Zahlen, Zeichenketten und booleschen Werten bleibt alles beim alten:

```
5
↪ 5
"Mike ist doof"
↪ "Mike ist doof"
#t
↪ #t
```

Bei Listen sieht es allerdings anders aus:

```
(list 1 2 3 4 5 6)
↪ (1 2 3 4 5 6)
```

Die REPL druckt also eine Liste aus, indem sie zuerst eine öffnende Klammer ausdrückt, dann die Listenelemente (durch Leerzeichen getrennt) und dann eine schließende Klammer.

Das funktioniert auch für die leere Liste:

```
empty
↪ ()
```

Mit der neuen Sprachebene bekommt außerdem der Apostroph, das dem Literal für die leere Liste voransteht, eine erweiterte Bedeutung. Unter anderem kann der Apostroph benutzt werden, um Literale für Listen zu formulieren:

```
'(1 2 3 4 5 6)
↪ (1 2 3 4 5 6)
'(1 #t "Mike" (2 3) "doof" 4 #f 17)
↪ (1 #t "Mike" (2 3) "doof" 4 #f 17)
'()
↪ ()
```

In der neuen Sprachebene benutzen die Literale und die ausgedruckten externen Repräsentationen für Listen also die gleiche Notation. Sie unterscheiden sich nur dadurch, daß beim Literal der Apostroph voransteht. Der Apostroph funktioniert auch bei Zahlen, Zeichenketten und booleschen Werten:

```
'5
↪ 5
'"Mike ist doof"
↪ "Mike ist doof"
'#t
↪ #t
```

Der Apostroph am Anfang eines Ausdrucks kennzeichnet diesen also als Literal. Der Wert des Literals wird genauso ausgedruckt, wie es im Programm steht. (Abgesehen von Leerzeichen und Zeilenumbrüchen.) Der Apostroph heißt auf englisch „quote“, und deshalb ist diese Literalschreibweise auch unter diesem Namen bekannt. Bei Zahlen, Zeichenketten und booleschen Literalen ist auch ohne Quote klar, daß es sich um Literale handelt. Das Quote ist darum bei ihnen rein optional; sie heißen *selbstquotierend*. Bei Listen hingegen sind Mißverständnisse mit anderen zusammengesetzten Formen möglich, die ja auch mit einer öffnenden Klammer beginnen: ¹

```
(1 2 3 4 5 6)
↪ procedure application: expected procedure, given: 1;
   arguments were: 2 3 4 5 6
```

Mit der Einführung von Quote kommt noch eine völlig neue Sorte Werte hinzu: die *Symbole*. Symbole sind Werte ähnlich wie Zeichenketten und bestehen aus Text. Sie unterscheiden sich allerdings dadurch, daß sie als Literal mit Quote geschrieben und in der REPL ohne Anführungszeichen ausgedruckt werden:

```
'mike
↪ mike
'doof
↪ doof
```

Symbole lassen sich mit dem Prädikat `symbol?` von anderen Werten unterscheiden:

```
(symbol? 'mike)
↪ #t
(symbol? 5)
↪ #f
(symbol? "Mike")
↪ #f
```

Vergleichen lassen sich Symbole mit `equal?` (siehe Abbildung 13.3):

¹Tatsächlich ist die neue Schreibweise für externe Repräsentationen die Standard-Repräsentation in Scheme. Die früheren Sprachebenen benutzten die alternative Schreibweise, um die Verwirrung zwischen Listenliteralen und zusammengesetzten Formen zu vermeiden.

```
(equal? 'mike 'herb)
↪ #f
(equal? 'mike 'mike)
↪ #t
```

Symbole können nicht aus beliebigem Text bestehen. Leerzeichen sind zum Beispiel verboten. Tatsächlich entsprechen die Namen der zulässigen Symbole genau den Namen von Variablen:

```
'karl-otto
↪ karl-otto
'mehrwertsteuer
↪ mehrwertsteuer
'duftmarke
↪ duftmarke
'lambda
↪ lambda
'+
↪ +
'*
↪ *
```

Diese Entsprechung wird in diesem Kapitel noch eine entscheidene Rolle spielen. Symbole können natürlich auch in Listen und damit auch in Listenliteralen vorkommen:

```
'(karl-otto mehrwertsteuer duftmarke)
↪ (karl-otto mehrwertsteuer duftmarke)
```

Mit Hilfe von Symbolen können Werte konstruiert werden, die in der REPL ausgedruckt wie Scheme-Ausdrücke aussehen:

```
'(+ 1 2)
↪ (+ 1 2)
'(lambda (n) (+ n 1))
↪ (lambda (n) (+ n 1))
```

Auch wenn diese Werte wie Ausdrücke so aussehen, sind sie doch ganz normale Listen: der Wert von `'(+ 1 2)` ist eine Liste mit drei Elementen: das Symbol `+`, die Zahl `1` und die Zahl `2`. Der Wert von `'(lambda (n) (+ n 1))` ist ebenfalls eine Liste mit drei Elementen: das Symbol `lambda`, eine Liste mit einem einzelnen Element, nämlich dem Symbol `n`, und einer weiteren Liste mit drei Elementen: dem Symbol `+`, dem Symbol `n` und der Zahl `1`.

Quote hat noch eine weitere verwirrende Eigenheit:

```
''()
↪ '()
```

Dieses Literal bezeichnet nicht die leere Liste (dann würde nur `()` ausgedruckt, ohne Quote), sondern etwas anderes:

```
(pair? ' '())
↪ #t
(first ' '())
↪ quote
(rest ' '())
↪ (())
```

Der Wert des Ausdrucks `' '()` ist also eine Liste mit zwei Elementen: das erste Element ist das Symbol `quote` und das zweite Element ist die leere Liste. `'t` ist selbst also nur syntaktischer Zucker, und zwar für `(quote t)`:

```
(equal? (quote ()) '())
↪ #t
(equal? (quote (quote ())) ' '())
↪ #t
```

Quote erlaubt die Konstruktion von Literalen für viele Werte, aber nicht für alle. Ein Wert, für den Quote ein Literal konstruieren kann, heißt *repräsentierbarer Wert*. Die folgende induktive Definition spezifiziert, was ein repräsentierbarer Wert ist:

- Zahlen, boolesche Werte, Zeichenketten und Symbole sind repräsentierbare Werte.
- Eine Liste aus repräsentierbaren Werten ist ihrerseits ein repräsentierbarer Wert.
- Nichts sonst ist ein repräsentierbarer Wert.

17.4 Implementierung der SECD-Maschine

Die SECD-Maschine ist ein Modell für die Implementierung des λ -Kalküls. Eine solche Implementierung läßt sich in Scheme einfach bauen – dieser Abschnitt zeigt, wie. Der grobe Fahrplan ergibt sich dabei aus der Struktur der SECD-Maschine selbst: Nach den obligatorischen Datendefinitionen müssen zunächst Terme in Maschinencode übersetzt werden. Dann kommt die Zustandsübergangsfunktion und schließlich die Auswertungsfunktion an die Reihe.

17.4.1 Datenanalyse

Die erste Aufgabe ist dabei zunächst, wie immer, die Datenanalyse: Am Anfang stehen die Terme des angewandten λ -Kalküls. Eine geeignete Repräsentation mit Listen und Symbolen läßt dabei die Terme in der „fortgeschrittenen“ Sprachebene genau wie entsprechenden Scheme-Terme aussehen:

<code>(+ 1 2)</code>	steht für	<code>(+ 1 2)</code>
<code>(lambda (x) x)</code>	steht für	<code>$\lambda x.x$</code>
<code>((lambda (x) (x x)) (lambda (x) (x x)))</code>	steht für	<code>$(\lambda x.(x x)) (\lambda x.(x x))$</code>
etc.		

Die Datendefinition dafür orientiert sich direkt an Definition 17.1:

Ein Lambda-Term ist eins der folgenden:

```
; - ein Symbol (für eine Variable)
; - eine zweielementige Liste (für eine reguläre Applikation)
; - eine Liste der Form (lambda (x) e) (für eine Abstraktion)
; - ein Basiswert
; - eine Liste mit einem Primitiv als erstem Element
;   (für eine primitive Applikation)
```

Hier die dazu passende Vertragsdefinition:

```
(define term
  (contract
    (mixed symbol
      application
      abstraction
      base
      primitive-application)))
```

Die Verträge für application etc. müssen noch definiert werden.

Um Verzweigungen über die Sorte term zu ermöglichen, müssen Prädikate für die einzelnen Teilsorten geschrieben werden. Diese können dann für die Definition der entsprechenden Verträge benutzt werden.

```
(: application? (%a -> boolean))
(define application?
  (lambda (t)
    (and (pair? t)
         (not (equal? 'lambda (first t)))
         (not (primitive? (first t))))))

(define application (contract (predicate application?)))

; Prädikat für Abstraktionen
(: abstraction? (%a -> boolean))
(define abstraction?
  (lambda (t)
    (and (pair? t)
         (equal? 'lambda (first t)))))

(define abstraction (contract (predicate abstraction?)))

; Prädikat für primitive Applikationen
(: primitive-application? (%a -> boolean))
(define primitive-application?
  (lambda (t)
    (and (pair? t)
```

```
(primitive? (first t))))))
```

```
(define primitive-application (contract (predicate primitive-application?)))
```

Die Definition läßt noch offen, was genau ein „Basiswert“ und was ein „Primitiv“ ist. Auch hierfür werden noch Datendefinitionen benötigt, zuerst für Basiswerte. Der Einfachheit halber beschränkt sich die Implementierung erst einmal auf boolesche Werte und Zahlen:

```
; Ein Basiswert ist ein boolescher Wert oder eine Zahl
```

Damit Basiswerte in Fallunterscheidungen von den anderen Arten von Termen unterschieden werden können, wird ein Prädikat benötigt:

```
; Prädikat für Basiswerte
(: base? (%a -> boolean))
(define base?
  (lambda (v)
    (or (boolean? v) (number? v))))
```

```
(define base (contract (predicate base?)))
```

Als nächstes sind Primitive gefragt: Am obigen Beispiel ist zu erkennen, daß z.B. + ein Primitiv sein sollte. Die Datendefinition für eine kleine beispielhafte Menge von Primitiven ist wie folgt:

```
; Ein Primitiv ist eins der Symbole +, -, *, /, =
```

Da die Primitive genau wie die Variablen Symbole sind, stehen die Primitive als Variablen nicht mehr zur Verfügung: Alle Symbole, die keine Primitive sind, sind also Variablen. Das dazugehörige Prädikat ist das folgende:

```
; Prädikat für Primitive
(: primitive? (%a -> boolean))
(define primitive?
  (lambda (s)
    (or (equal? '+ s)
        (equal? '- s)
        (equal? '* s)
        (equal? '/ s)
        (equal? '= s))))
```

```
(define primitive (contract (predicate primitive?)))
```

Bevor nun ein die SECD-Maschine einen Term verarbeiten kann, muß dieser erst in Maschinencode übersetzt werden. Dabei entsteht aus Definition 17.4 direkt Daten- und Vertragsdefinitionen für Instruktionen und Maschinencode:

```
; Eine Instruktion ist eins der folgenden:
```

```

; - ein Basiswert
; - eine Variable
; - eine Applikations-Instruktion
; - eine Instruktion für eine primitive Applikation
; - eine Abstraktion
(define instruction
  (contract
    (mixed base
      symbol
      ap
      tailap
      prim
      abs))

```

```

; Eine Maschinencode-Programm ist eine Liste von Instruktionen.
(define machine-code (contract (list instruction)))

```

Bei der Definition von Instruktionen ist wieder einiges Wunschenken im Spiel. Basiswerte und Variablen sind wie bei den Termen. Die restlichen Fälle werden durch eigene Daten-Definitionen abgebildet. Wie schon bei den leeren Bäumen sind Record-Definitionen ohne Felder im Spiel, die Fallunterscheidungen möglich machen:

```

; Eine Applikations-Instruktion ist ein Wert
; (make-ap)
(define-record-procedures ap
  make-ap ap?
  ())
(: make-ap (-> ap))

; Die Instruktion für eine primitive Applikation
; ist ein Wert
; (real-make-prim op arity)
; wobei op ein Symbol und arity die Stelligkeit
; ist
(define-record-procedures prim
  real-make-prim prim?
  (prim-operator prim-arity))
(: make-prim (symbol natural -> prim))

; Eine Abstraktions-Instruktion ist ein Wert
; (make-abs v c)
; wobei v ein Symbol (für eine Variable) und c
; Maschinencode ist
(define-record-procedures abs
  make-abs abs?
  (abs-variable abs-code))

```

```
(: make-abs (symbol machine-code -> abs))
```

Da die Stelligkeit eines Primitivs dem Primitiv fest zugeordnet ist, ist eine Hilfsprozedur nützlich, die bei der Erzeugung eines Werts der Sorte `prim` die Stelligkeit ergänzt. Glücklicherweise haben alle oben eingeführten Primitive die gleiche Stelligkeit:

```
; Primitiv erzeugen
(: make-prim (symbol -> prim))
(define make-prim
  (lambda (s)
    (real-make-prim s 2)))
```

Die Einführung von Primitive mit anderen Stelligkeiten ist Gegenstand von Aufgabe 17.6.

17.4.2 Übersetzung in Maschinencode

Nun, da sowohl Terme als auch der Maschinencode Datendefinitionen haben, ist es möglich, die Übersetzung zu programmieren. Hier sind Kurzbeschreibung, Vertrag und Gerüst:

```
; Term in Maschinencode übersetzen
(: term->machine-code (term -> machine-code))
(define term->machine-code
  (lambda (e)
    ...))
```

Da es sich bei `term` um gemischte Daten handelt, muß – wie immer – eine Verzweigung den Rumpf der Prozedur bilden:

```
(define term->machine-code
  (lambda (e)
    (cond
      ((symbol? e) ...)
      ((application? e) ...)
      ((abstraction? e) ...)
      ((base? e) ...)
      ((primitive-application? e) ...))))
```

Die Implementierung entspricht in den einzelnen Fällen genau der Übersetzungsfunktion `[[]]`. Die Fälle für Variablen und Basiswerte sind, genau wie dort, trivial:

```
(define term->machine-code
  (lambda (e)
    (cond
      ((symbol? e) (list e))
      ((base? e) (list e))
      ...)))
```

Bei regulären Applikationen werden Operator und Operand übersetzt, und das ganze zusammen mit einer `ap`-Instruktion zu einer Liste zusammengesetzt:

```
(define term->machine-code
  (lambda (e)
    (cond
      ...
      ((application? e)
       (append (term->machine-code (first e))
                (append (term->machine-code (first (rest e)))
                        (list (make-ap))))))
      ...)))
```

Bei den primitiven Applikationen werden erst einmal die Operanden in Maschinencode übersetzt, die Resultate aneinandergehängt, und schließlich kommt noch eine prim-Instruktion ans Ende:

```
(define term->machine-code
  (lambda (e)
    (cond
      ...
      ((primitive-application? e)
       (append
        (append-lists
         (map term->machine-code (rest e)))
         (list (make-prim (first e))))))
      ...)))
```

Dieses Stück Code benutzt die Hilfsprozedur `append-lists`, die aus einer Liste von Listen eine einzelne Liste macht, indem die Elemente aneinandergehängt werden:

```
; die Elemente einer Liste von Listen aneinanderhängen
(: append-lists ((list (list %a)) -> (list %a)))
(define append-lists
  (lambda (l)
    (cond
      ((empty? l) empty)
      ((pair? l)
       (append (first l) (concatenate (rest l)))))))
```

Zurück zur Übersetzung: Eine Abstraktionen wird direkt in eine `abs`-Instruktion übersetzt, wobei der Rumpf selbst noch in Maschinencode übersetzt wird:

```
(define term->machine-code
  (lambda (e)
    (cond
      ...
      ((abstraction? e)
       (list
        (make-abs (first (first (rest e)))
```

```
(term->machine-code
 (first (rest (rest e)))))))))
```

17.4.3 Zustandsübergang und Auswertung

Da nun alle λ -Terme in Maschinencode-Programme übersetzt werden können, ist jetzt die eigentliche SECD-Maschine an der Reihe. Hier sind erst einmal einige neue Datendefinitionen fällig. Zunächst einmal die Menge S der Stacks:

```
; Ein Stack ist eine Liste von Werten
(define stack (contract (list value)))
```

Die Definition von Werten W kommt etwas später an die Reihe.

Umgebungen aus der Menge E sind mathematisch gesehen Mengen aus Tupeln. In der Implementierung werden sie dargestellt aus Listen von *Bindungen*, wobei jede Bindung einem Tupel aus der mathematischen Definition entspricht:

```
; Eine Umgebung ist eine Liste von Bindungen.
; Dabei gibt es für jede Variable nur eine Bindung.
(define environment (contract (list binding)))

; Eine Bindung (Name: binding) ist ein Wert
; (make-binding v x)
; wobei v der Name einer Variablen und x der dazugehörige Wert ist.
```

```
(define-record-procedures binding
  make-binding binding?
  (binding-variable binding-value))
(: make-binding (symbol value -> binding))
```

Die leere Umgebung wird öfter benötigt und wird darum schon vordefiniert:

```
; die leere Umgebung
(define the-empty-environment empty)
```

Zwei Operationen gibt es für eine Umgebung e : die Erweiterung um eine Bindung $e[v \mapsto w]$ und das Nachschauen einer Bindung $e(v)$. Zunächst die Erweiterung: die Implementierung entspricht genau der mathematischen Definition: zunächst wird eine eventuell vorhandene Bindung für v entfernt, dann eine neue Bindung hinzugefügt:

```
; eine Umgebung um eine Bindung erweitern
(: extend-environment (environment symbol value -> environment))
(define extend-environment
  (lambda (e v w)
    (make-pair (make-binding v w)
              (remove-environment-binding e v))))
```

Für das Entfernen der alten Bindung ist die Hilfsprozedur `remove-environment-binding` zuständig. Sie folgt einmal mehr strikt der Konstruktionsanleitung für Prozeduren, die Listen konsumieren:

```
; die Bindung für eine Variable aus einer Umgebung entfernen
(: remove-environment-binding (environment symbol -> environment))
(define remove-environment-binding
  (lambda (e v)
    (cond
      ((empty? e) empty)
      ((pair? e)
       (if (equal? v (binding-variable (first e)))
           (rest e)
           (make-pair (first e)
                      (remove-environment-binding (rest e) v)))))))
```

Auch die zweite Operation, das Nachschauen einer Bindung in der Umgebung, folgt der Konstruktionsanleitung:

```
; die Bindung für eine Variable in einer Umgebung finden
(: lookup-environment (environment symbol -> value))
(define lookup-environment
  (lambda (e v)
    (cond
      ((empty? e) (violation "unbound variable"))
      ((pair? e)
       (if (equal? v (binding-variable (first e)))
           (binding-value (first e))
           (lookup-environment (rest e) v))))))
```

Damit sind die Operationen auf Umgebungen abgeschlossen. Als nächstes sind Dumps an der Reihe: D ist als Folge von Tupeln $S \times E \times C$ definiert, auch genannt *Frames*. Hier sind Daten- und Record-Definition:

```
; Ein Dump ist eine Liste von Frames

; Ein Frame ist ein Wert
; (make-frame s e c)
; wobei s ein Stack, e eine Umgebung und c Maschinencode ist.
(define-record-procedures frame
  make-frame frame?
  (frame-stack frame-environment frame-code))
(: make-frame (stack environment machine-code -> frame))
```

Schließlich fehlt noch eine Repräsentation für die Menge W der Werte: Ein Wert ist entweder ein Basiswert oder eine Closure. Basiswerte wurden bereits in Abschnitt 17.4.1 definiert; es fehlen noch Closures, die Tupel aus $V \times C \times E$ sind. Hier sind die entsprechenden Definitionen:

```

; Ein SECD-Wert ist ein Basiswert oder eine Closure
(define value (contract (mixed base closure)))

; Eine Closure ist ein Wert
; (make-closure v c e)
; wobei v die Variable der Lambda-Abstraktion,
; c der Code der Lambda-Abstraktion
; und e ein Environment ist.
(define-record-procedures closure
  make-closure closure?
  (closure-variable closure-code closure-environment))
(: make-closure (symbol machine-code environment -> closure))

```

Mit Hilfe dieser Definitionen ist es möglich, eine Daten- und eine Record-Definition für die Zustände der SECD-Maschine anzugeben, also die Tupel aus $S \times E \times C \times D$:

```

; Ein SECD-Zustand ist ein Wert
; (make-secd s e c d)
; wobei s ein Stack, e eine Umgebung, c Maschinencode
; und d ein Dump ist
(define-record-procedures secd
  make-secd secd?
  (secd-stack secd-environment secd-code secd-dump))
(: make-secd (stack environment machine-code dump -> secd))

```

Damit kann es an die Zustandsübergangsfunktion gehen. Sie wird als Prozedur realisiert, die einen SECD-Zustand konsumiert und einen neuen liefert. Hier sind Kurzbeschreibung, Vertrag und Gerüst:

```

; Zustandsübergang berechnen
(: secd-step (secd -> secd))
(define secd-step
  (lambda (state)
    ...))

```

Entsprechend den Regeln der SECD-Maschine muß der Rumpf der Prozedur eine Verzeigung zwischen den verschiedenen Fällen bei der Code-Komponente von state sein. Diese folgen den Konstruktionsanleitungen für Listen und für gemischte Daten. Es ist bereits an den Regeln abzulesen, daß alle Regeln Zugriff auf die Komponenten von state benötigen. Für diese werden gleich am Anfang lokale Variablen angelegt:

```

(define secd-step
  (lambda (state)
    (let ((stack (secd-stack state))
          (environment (secd-environment state))
          (code (secd-code state))
          (dump (secd-dump state)))

```

```
(cond
  ((pair? code)
   (cond
    ((base? (first code)) ...)
    ((symbol? (first code)) ...)
    ((prim? (first code)) ...)
    ((abs? (first code)) ...)
    ((ap? (first code)) ...)))
  ((empty? code) ...))))
```

In diesem Gerüst werden nun die Regeln direkt abgebildet. Hier zur Erinnerung noch einmal die erste Regel für Basiswerte:

$$(\underline{s}, e, b\underline{c}, \underline{d}) \leftrightarrow (b\underline{s}, e, \underline{c}, \underline{d})$$

Hier der passende Code dafür:

```
(define secd-step
  (lambda (state)
    ...
    (cond
      ((base? (first code))
       (make-secd (make-pair (first code) stack)
                  environment
                  (rest code)
                  dump))
      ...))
    ...))
```

Hier die Regel für Variablen:

$$(\underline{s}, e, v\underline{c}, \underline{d}) \leftrightarrow (e(v)\underline{s}, e, \underline{c}, \underline{d})$$

Hier der entsprechende Code:

```
(define secd-step
  (lambda (state)
    ...
    (cond
      ((symbol? (first code))
       (make-secd (make-pair
                  (lookup-environment environment (first code))
                  stack)
                  environment
                  (rest code)
                  dump))
      ...))
    ...))
```

Die Regel für primitive Applikationen ist etwas aufwendiger:

$$(b_n \dots b_1 \underline{s}, e, \text{prim}_{F\underline{c}}, \underline{d}) \hookrightarrow (b\underline{s}, e, \underline{c}, \underline{d})$$

wobei $F \in \Sigma^{(n)}$ und $F_B(b_1, \dots, b_n) = b$

Für die Implementierung werden Hilfsprozeduren gebraucht, welche die Argumente vom Stack holen und in der Reihenfolge umdrehen, die Argumente vom Stack entfernen und schließlich die eigentliche δ -Transition berechnen:

```
(define secd-step
  (lambda (state)
    ...
    (cond
      ...
      ((prim? (first code))
       (make-secd (make-pair
                   (apply-primitive
                    (prim-operator (first code))
                    (take-reverse (prim-arity (first code)) stack))
                  (drop (prim-arity (first code)) stack))
                 environment
                 (rest code)
                 dump))
       ...))
    ...))
```

Die Prozedur drop ist gerade die in Aufgabe 7.8 geforderte Prozedur:

```
; die ersten Elemente einer Liste weglassen
(: drop (natural (list %a) -> (list %a)))
```

Die take-reverse-Prozedur ist das Pendant zu drop, das die ersten n Elemente einer Liste in umgekehrter Reihenfolge liefert. Dies ist am einfachsten über eine endrekursive Hilfsprozedur zu erledigen – aus Kapitel 7 ist ja bekannt, daß bei endrekursiver Konstruktion von Listen gerade immer die Reihenfolge umgedreht wird:

```
; die ersten Elemente einer Liste in umgekehrter Reihenfolge berechnen
(: take-reverse (natural (list %a) -> (list %a)))
(define take-reverse
  (lambda (n l)
    ; (: loop (natural (list a) (list a) -> (list a)))
    (letrec ((loop (lambda (n l r)
                     (if (= n 0)
                         r
                         (loop (- n 1) (rest l) (make-pair (first l) r))))))
      (loop n l '()))))
```

Aus einem Primitiv und einer Liste von Argumenten berechnet `apply-primitive` das Resultat der primitiven Applikation. Dabei handelt es sich bei `primitive` um eine Fallunterscheidung, der Rumpf der Prozedur ist also eine entsprechende Verzweigung:

```
; Delta-Transition berechnen
(: apply-primitive (primitive (list value) -> value))
(define apply-primitive
  (lambda (p args)
    (cond
      ((equal? p '+)
       (+ (first args) (first (rest args))))
      ((equal? p '-')
       (- (first args) (first (rest args))))
      ((equal? p '=)
       (= (first args) (first (rest args))))
      ((equal? p '*')
       (* (first args) (first (rest args))))
      ((equal? p '/')
       (/ (first args) (first (rest args)))))))
```

Die Regel für Abstraktionen macht aus einer Abstraktion eine Closure:

$$(\underline{s}, e, (v, \underline{c}') \underline{c}, \underline{d}) \hookrightarrow ((v, \underline{c}', e) \underline{s}, e, \underline{c}, \underline{d})$$

Der Code macht dies genauso:

```
(define secd-step
  (lambda (state)
    ...
    (cond
      ...
      ((abs? (first code))
       (make-secd (make-pair
                   (make-closure (abs-variable (first code))
                                  (abs-code (first code))
                                  environment)
                   stack)
              environment
              (rest code)
              dump)))
    ...)))
```

Hier die Regel für die Applikation:

$$(w(v, \underline{c}', e') \underline{s}, e, \text{ap } \underline{c}, \underline{d}) \hookrightarrow (\epsilon, e'[v \mapsto w], \underline{c}', (\underline{s}, e, \underline{c}) \underline{d})$$

Hier der Code dazu:

```
(define secd-step
```

```

(lambda (state)
  ...
  (cond
    ...
    ((ap? (first code))
     (let ((closure (first (rest stack))))
       (make-secd empty
                  (extend-environment
                   (closure-environment closure)
                   (closure-variable closure)
                   (first stack))
                  (closure-code closure)
                  (make-pair
                   (make-frame (rest (rest stack))
                               environment (rest code))
                   dump))))
    ...))
  ...))

```

Schließlich bleibt noch der Code für die Rückgabe eines Wertes von einer Prozedur. Hier ist die Regel:

$$(w, e, \epsilon, (\underline{s'}, e', \underline{c'}) \underline{d}) \leftrightarrow (w \underline{s'}, e', \underline{c'}, \underline{d})$$

Hier ist der Code dazu:

```

(define secd-step
  (lambda (state)
    ...
    (cond
      ...
      ((empty? code)
       (let ((f (first dump)))
         (make-secd
          (make-pair (first stack)
                    (frame-stack f))
          (frame-environment f)
          (frame-code f)
          (rest dump))))
      ...))
  ...))

```

Damit die SECD-Maschine in Betrieb genommen werden kann, muß ein Term e noch in einen Anfangszustand $(\epsilon, \emptyset, \llbracket e \rrbracket, \epsilon)$ übersetzt werden. Das erledigt folgende Hilfsprozedur:

```

; Aus Term SECD-Anfangszustand machen
(: inject-secd (term -> secd))
(define inject-secd
  (lambda (e)

```

```
(make-secd empty
  the-empty-environment
  (term->machine-code e)
  empty))
```

Damit läßt sich die Maschine schon ausprobieren:

```
(secd-step (inject-secd '(+ 1 2)))
↪#<record:secd (1) () (2 #<record:prim + 2>) ()>
(secd-step (secd-step (inject-secd '(+ 1 2))))
↪#<record:secd (2 1) () (#<record:prim + 2>) ()>
(secd-step (secd-step (secd-step (inject-secd '(+ 1 2)))))
↪#<record:secd (3) () () ()>
```

Es fehlt noch die Auswertungsfunktion $\text{eval}_{\text{SECD}}$, die eine Hilfsprozedur benötigt, um die reflexiv-transitive Hülle des Zustandsübergangs \hookrightarrow^* benötigt:

```
; bis zum Ende Zustandsübergänge berechnen
(: secd-step* (secd -> secd))
(define secd-step*
  (lambda (state)
    (if (and (empty? (secd-code state))
             (empty? (secd-dump state)))
        state
        (secd-step* (secd-step state)))))
```

Die Auswertungsfunktion orientiert sich direkt an der mathematischen Definition:

```
; Evaluationsfunktion zur SECD-Maschine berechnen
(: eval-secd (term -> (mixed value (one-of 'function))))
(define eval-secd
  (lambda (e)
    (let ((val (first
                 (secd-stack
                  (secd-step*
                   (inject-secd e))))))
      (if (base? val)
          val
          'proc))))
```

Damit läuft die SECD-Maschine:

```
(eval-secd '(((lambda (x) (lambda (y) (+ x y))) 1) 2))
↪3
```

17.5 Die endrekursive SECD-Maschine

Die SECD-Maschine hat einen Schönheitsfehler: Bei endkursiven Applikationen sollte sie eigentlich, wie in Scheme, keinerlei zusätzlichen Platz verbrauchen, da kein Kontext anfällt. Folgende Beispielauswertung für den Term $(\lambda x.x x)$ zeigt aber, daß der Zustand mit fortschreitender Auswertung immer größer wird:

$(\epsilon,$	$\emptyset,$	$(x, x x \text{ ap}) (x, x x \text{ ap}) \text{ ap},$	$\epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset),$	$\emptyset,$	$(x, x x \text{ ap}) \text{ ap},$	$\epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset) (x, x x \text{ ap}, \emptyset),$	$\emptyset,$	$\text{ap},$	$\epsilon)$
$\hookrightarrow (\epsilon,$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x x \text{ ap},$	$(\epsilon, \emptyset, \epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset),$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x \text{ ap},$	$(\epsilon, \emptyset, \epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset) (x, x x \text{ ap}, \emptyset),$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$\text{ap},$	$(\epsilon, \emptyset, \epsilon)$
$\hookrightarrow (\epsilon,$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x x \text{ ap},$	$(\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \emptyset, \epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset),$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x \text{ ap},$	$(\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \emptyset, \epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset) (x, x x \text{ ap}, \emptyset),$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$\text{ap},$	$(\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \emptyset, \epsilon)$
$\hookrightarrow (\epsilon,$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x x \text{ ap},$	$(\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \emptyset, \epsilon)$
$\hookrightarrow ((x, x x \text{ ap}, \emptyset),$	$\{(x, (x, x x \text{ ap}, \emptyset))\},$	$x \text{ ap},$	$(\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \{(x, (x, x x \text{ ap}, \emptyset))\}, \epsilon) (\epsilon, \emptyset, \epsilon)$
...			

Damit ist die SECD-Maschine, so wie ist, als Ausführungsmodell für Scheme ungeeignet. Dieses Manko läßt sich zum Glück reparieren: Die SECD-Maschine muß endrekursive und „normale“ Applikationen unterschiedlich behandeln. Dazu wird eine neue Instruktion namens `tailap` eingeführt, die wie `ap` eine Applikation durchführt, aber eine endrekursive Applikation signalisiert:

$$I = \dots \cup \{\text{tailap}\}$$

Als nächstes muß die Übersetzungsfunktion von Termen in Maschinencode geändert werden: Applikationen, die Kontext um sich herum haben, sollen mit `ap` übersetzt werden, solche ohne Kontext mit `tailap`. Da der Applikation allein der Kontext nicht anzusehen ist, sondern nur dem Term „drumherum“, wird die Übersetzungsfunktion $\llbracket _ \rrbracket$ in zwei Teile aufgespalten: für einen Term e wird die Auswertungsfunktion $\llbracket _ \rrbracket$ immer dann benutzt, wenn um e Kontext steht. Eine weitere Funktion $\llbracket _ \rrbracket'$ wird immer dann aufgerufen, wenn *kein* Kontext drumherum steht.

Kontext entsteht seinerseits immer durch Funktionsapplikationen. Bei der Auswertung eines Terms $(e_0 e_1)$ muß *nach* e_0 noch e_1 ausgewertet werden, und nach Auswertung von e_1 muß noch die Applikation durchgeführt werden. Sowohl e_0 als auch e_1 stehen in Kontext. Ähnlich ist es bei den Argumenten von primitiven Applikationen.

Auf der anderen Seite schneiden Abstraktionen für ihren Rumpf den Kontext erst einmal ab: Der Rumpf einer Abstraktion kommt schließlich bei der Auswertung der Abstraktion noch gar nicht zum Zug. Ob er Kontext hat oder nicht, entscheidet sich erst bei der Applikation. Dementsprechend schalten Applikationen und Abstraktionen zwischen den beiden

Funktionen $\llbracket _ \rrbracket$ und $\llbracket _ \rrbracket'$ hin und her:

$$\begin{aligned} \llbracket _ \rrbracket & : \mathcal{L}_{\lambda A} \rightarrow C \\ \llbracket e \rrbracket & \stackrel{\text{def}}{=} \begin{cases} b & \text{falls } e = b \in B \\ v & \text{falls } e = v \in V \\ \llbracket e_0 \rrbracket \llbracket e_1 \rrbracket \text{ ap} & \text{falls } e = (e_0 e_1) \\ \llbracket e_1 \rrbracket \dots \llbracket e_n \rrbracket \text{ prim}_F & \text{falls } e = (F e_1 \dots e_n) \\ (v, \llbracket e_0 \rrbracket') & \text{falls } e = \lambda v. e_0 \end{cases} \\ \llbracket _ \rrbracket' & : \mathcal{L}_{\lambda A} \rightarrow C \\ \llbracket e \rrbracket' & \stackrel{\text{def}}{=} \begin{cases} b & \text{falls } e = b \in B \\ v & \text{falls } e = v \in V \\ \llbracket e_0 \rrbracket \llbracket e_1 \rrbracket \text{ tailap} & \text{falls } e = (e_0 e_1) \\ \llbracket e_1 \rrbracket \dots \llbracket e_n \rrbracket \text{ prim}_F & \text{falls } e = (F e_1 \dots e_n) \\ (v, \llbracket e_0 \rrbracket') & \text{falls } e = \lambda v. e_0 \end{cases} \end{aligned}$$

Die Übersetzungsfunktion hat die eigentliche Arbeit geleistet: Jetzt muß nur noch eine Zustandsübergangsregel her, die `tailap` verarbeitet. Diese ergibt sich direkt aus den Regeln für `ap` und die Rückgabe eines Wertes: `tailap` funktioniert so, wie `ap` direkt gefolgt von der Rückgaberegeln. Hier sind die beiden Regeln noch einmal zur Erinnerung:

$$\begin{aligned} (w(v, \underline{c}', e') \underline{s}, e, \text{ap } \underline{c}, \underline{d}) & \hookrightarrow (\varepsilon, e'[v \mapsto w], \underline{c}', (\underline{s}, e, \underline{c}) \underline{d}) \\ (w, e, \varepsilon, (\underline{s}', e', \underline{c}') \underline{d}) & \hookrightarrow (w \underline{s}', e', \underline{c}', \underline{d}) \end{aligned}$$

Da die erste Regel ein neues Dump-Frame erzeugt und die zweite ein Dump-Frame „vernichtet“, entfällt diese Arbeit in der Regel für `tailap`:

$$(w(v, \underline{c}', e') \underline{s}, e, \text{tailap } \underline{c}, \underline{d}) \hookrightarrow (\underline{s}, e'[v \mapsto w], \underline{c}', \underline{d})$$

Damit läuft das Beispiel zwar immer noch endlos, aber immerhin, ohne immer mehr Platz zu verbrauchen:

$$\begin{array}{llll} \hookrightarrow (\varepsilon, & \emptyset, & (x, x x \text{ tailap}) (x, x x \text{ tailap}) \text{ ap}, & \varepsilon) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset), & \emptyset, & (x, x x \text{ tailap}) \text{ ap}, & \varepsilon) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset) (x, x x \text{ tailap}, \emptyset), & \emptyset, & \text{ap}, & \varepsilon) \\ \hookrightarrow (\varepsilon, & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset) (x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & \text{tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow (\varepsilon, & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset) (x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & \text{tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow (\varepsilon, & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset) (x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & \text{tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow (\varepsilon, & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \\ \hookrightarrow ((x, x x \text{ tailap}, \emptyset), & \{(x, (x, x x \text{ tailap}, \emptyset))\}, & x \text{ tailap}, & (\varepsilon, \emptyset, \varepsilon)) \end{array}$$

Die Implementierung der endrekursiven SECD-Maschine ist Gegenstand von Übungsaufgabe 17.3.

17.6 Der λ -Kalkül mit Zustand

Der bisher vorgestellte λ -Kalkül liefert keinerlei Erklärung für das Verhalten von Zuweisungen. Tatsächlich hat sich schon in Abschnitt 13.4 angedeutet, daß Zuweisungen die Formalisierung deutlich erschweren. Möglich ist es trotzdem, und dieser Abschnitt zeigt, wie es geht.

Als erstes muß wieder einmal die Sprache des λ -Kalküls erweitert werden, diesmal um `set!`-Ausdrücke:

Definition 17.5 (Sprache des angewandten λ -Kalküls mit Zustand $\mathcal{L}_{\lambda S}$) Sei V eine abzählbare Menge von Variablen. Sei A eine abzählbare Menge von Adressen mit $V \cap A = \emptyset$. Sei Σ ein Operationsalphabet mit `void` $\in \Sigma^{(0)}$ und $(B, \{F_B \mid F \in \Sigma\})$ eine Σ -Algebra von Basiswerten. Die Sprache des angewandten λ -Kalküls mit Zustand, die Menge der *angewandten λ -Terme mit Zustand*, $\mathcal{L}_{\lambda S}$, ist die kleinste Menge mit folgenden Eigenschaften:

1. $V \subseteq \mathcal{L}_{\lambda S}$
2. Für $e_0, e_1 \in \mathcal{L}_{\lambda S}$ ist auch $(e_0 \ e_1) \in \mathcal{L}_{\lambda S}$.
3. Für $x \in V, e \in \mathcal{L}_{\lambda S}$ ist auch $(\lambda x.e) \in \mathcal{L}_{\lambda S}$.
4. $B \subseteq \mathcal{L}_{\lambda S}$
5. Für $e_1, \dots, e_n \in \mathcal{L}_{\lambda S}$ und $F \in \Sigma^{(n)}$ ist $(F \ e_1 \ \dots \ e_n) \in \mathcal{L}_{\lambda S}$.
6. Für $v \in V$ und $e \in \mathcal{L}_{\lambda S}$ ist $(\text{set! } v \ e) \in \mathcal{L}_{\lambda S}$.

Der `void`-Wert wird als Rückgabewert von `set!`-Ausdrücken dienen.

Um Reduktionsregeln für Zuweisungen zu bilden, ist es notwendig, den Begriff des *Speichers* in den λ -Kalkül einzuführen: Im λ -Kalkül mit Zustand stehen Variablen nicht mehr für Werte, die für sie eingesetzt werden können, sondern für *Speicherzellen*. Eine Speicherzelle ist ein Ort im Speicher, der einen Wert aufnimmt, der auch wieder verändert werden kann. Dabei wird jede Speicherzelle durch eine *Adresse* identifiziert. Eine Adresse ist eine abstrakte Größe, es kommt also gar nicht darauf an, um was für eine Art Wert es sich handelt – im realen Computer ist eine Adresse in der Regel einfach eine Zahl. In diesem Abschnitt steht A für die Menge der Adressen, die abzählbar sein sollte.

Ein Speicher aus der Menge M ist eine Zuordnung zwischen Adressen aus A und Werten. Die Werte sind wie schon im normalen λ -Kalkül die Basiswerte und die Abstraktionen – hier bekommen sie, weil sie eine Rolle in den Reduktionsregeln spielen, den Namen X :

$$\begin{aligned} M &= \mathcal{P}(A \times X) \\ X &= B \cup \{\lambda v.e \mid \lambda v.e \in \mathcal{L}_{\lambda S}\} \end{aligned}$$

Um Reduktionsregeln für den λ -Kalkül mit Zustand zu formulieren, muß $\mathcal{L}_{\lambda S}$ noch erweitert werden, damit die Adressen ins Spiel kommen: Adressen werden Terme und sind auf der linken Seite von Zuweisungen zulässig:

7. $A \subseteq \mathcal{L}_{\lambda S}$.

8. Für $a \in A$ und $e \in \mathcal{L}_{\lambda S}$ ist $(\text{set! } a \ e) \in \mathcal{L}_{\lambda S}$.

Adressen tauchen dabei nur als Zwischenschritte bei der Reduktion auf; sie sind nicht dafür gedacht, daß sie der Programmierer in ein Programm schreibt.

Da das bisherige Substitutionsprinzip bei Zuweisungen nicht mehr funktioniert, reicht es nicht, die Reduktionsregeln für den λ -Kalkül mit Zustand einfach nur auf Termen auszudrücken: Ein Term, der ja Adressen enthalten kann, ergibt nur Sinn, wenn er mit einem Speicher kombiniert wird. Die Reduktionsregeln überführen somit immer ein Paar, bestehend aus einem Term und einem Speicher in ein ebensolches Paar. Hierbei wird der Einfachheit halber kein Unterschied mehr zwischen den verschiedenen Arten der Reduktion gemacht:

$$\begin{aligned} b, m &\rightarrow a, m[a \mapsto b] \text{ wobei } a \text{ frisch} \\ \lambda v. e, m &\rightarrow a, m[a \mapsto \lambda v. e] \text{ wobei } a \text{ frisch} \\ (e_0 \ a_1), m &\rightarrow e[v \mapsto a], m[a \mapsto m(a_1)] \text{ wobei } m(a_0) = \lambda v. e \text{ und } a \text{ frisch} \\ (\text{set! } a_0 \ a_1), m &\rightarrow \text{void}, m[a_0 \mapsto m(a_1)] \\ (F a_1 \dots a_n), m &\rightarrow a, m[a \mapsto F_B(b_1, \dots, b_n)] \text{ wobei } b_i = m(a_i) \in B, a \text{ frisch} \end{aligned}$$

Die Formulierung „ a frisch“ bedeutet dabei, daß a eine Adresse sein sollte, die in m bisher noch nicht benutzt wurde. Die Operation $m[a \mapsto x]$ ist ähnlich wie bei Umgebungen definiert: der alte Speicherinhalt bei a wird zunächst entfernt, und dann eine neue Zuordnung für a nach x hinzugefügt:

$$m[a \mapsto x] \stackrel{\text{def}}{=} (e \setminus \{(a, x') \mid (a, x') \in m\}) \cup \{(a, x)\}$$

Die Regeln sind immer noch über Substitution definiert, allerdings werden für Variablen jetzt nicht mehr Werte sondern Adressen eingesetzt. Sie werden, wie beim normalen Call-by-Value-Kalkül auch, auf Subterme fortgesetzt, die möglichst weit links innen stehen.

Im folgenden Beispiel stehen fettgedruckte Zahlen **0**, **1** für Adressen. Die Redexe sind jeweils unterstrichen:

$$\begin{aligned} &((\lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1)))) \ 12), \emptyset \\ &\rightarrow (\mathbf{0} \ \underline{12}), \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\} \\ &\rightarrow (\mathbf{0} \ \mathbf{1}), \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12)\} \\ &\rightarrow ((\lambda y. \mathbf{2})(\text{set! } \mathbf{2} \ (+ \ \mathbf{2} \ 1))), \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 12)\} \\ &\rightarrow (\mathbf{3} \ (\text{set! } \mathbf{2} \ (+ \ \mathbf{2} \ \underline{1}))), \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 12), (\mathbf{3}, (\lambda y. \mathbf{2}))\} \\ &\rightarrow (\mathbf{3} \ (\text{set! } \mathbf{2} \ (+ \ \mathbf{2} \ \underline{\mathbf{4}}))), \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 12), (\mathbf{3}, (\lambda y. \mathbf{2})), (\mathbf{4}, 1)\} \\ &\rightarrow (\mathbf{3} \ (\text{set! } \mathbf{2} \ \underline{\mathbf{5}})), \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 12), (\mathbf{3}, (\lambda y. \mathbf{2})), (\mathbf{4}, 1), (\mathbf{5}, 13)\} \\ &\rightarrow (\mathbf{3} \ \text{void}), \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 13), (\mathbf{3}, (\lambda y. \mathbf{2})), (\mathbf{4}, 1), (\mathbf{5}, 13)\} \\ &\rightarrow (\mathbf{3} \ \mathbf{6}), \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 13), (\mathbf{3}, (\lambda y. \mathbf{2})), (\mathbf{4}, 1), (\mathbf{5}, 13), (\mathbf{6}, \text{void})\} \\ &\rightarrow \mathbf{2}, \\ &\quad \{(\mathbf{0}, \lambda x. ((\lambda y. x)(\text{set! } x \ (+ \ x \ 1))))\}, (\mathbf{1}, 12), (\mathbf{2}, 13), (\mathbf{3}, (\lambda y. \mathbf{2})), (\mathbf{4}, 1), (\mathbf{5}, 13), (\mathbf{6}, \text{void})\} \end{aligned}$$

Der Endausdruck steht für die Speicherzelle an Adresse **2**, wo der Wert 13 steht. Es

ist sichtbar, daß die Auswertungsmaschinerie durch die Einführung von Zustand deutlich komplizierter wird.

17.7 Die SECDH-Maschine

Die SECD-Maschine ist nicht mächtig genug, um den λ -Kalkül mit Zustand zu modellieren: Es fehlt ein Speicher. Darum muß das Maschinen-Pendant zum λ -Kalkül mit Zustand um eine Speicher-Komponente erweitert werden: Heraus kommt die *SECDH-Maschine*, um die es in diesem Abschnitt geht.

Der Maschinencode für die SECDH-Maschine ist dabei genau wie bei der SECD-Maschine, nur daß eine spezielle Zuweisungsoperation hinzukommt:

$$I = \dots \cup \{ := \}$$

Die Übersetzungsfunktion produziert diese neue Instruktion bei `set!`-Ausdrücken:

$$\llbracket e \rrbracket \stackrel{\text{def}}{=} \begin{cases} \dots \\ v \llbracket e' \rrbracket := & \text{falls } e = (\text{set! } v e') \end{cases}$$

Der Begriff der Adresse aus der Menge A wird direkt aus dem Kalkül übernommen. Ähnlich wie im Kalkül landen Zwischenergebnisse nicht mehr direkt auf dem Stack, sondern stattdessen landen ihre Adressen im Speicher. Dementsprechend bilden nun Umgebungen Variablen auf Adressen ab. Die neue Komponente H ist gerade der Speicher, auch genannt *Heap*, der die Adressen auf Werte abbildet:

$$\begin{aligned} S &= A^* \\ E &= \mathcal{P}(V \times A) \\ D &= (S \times E \times C \times D)^* \\ H &= \mathcal{P}(A \times W) \\ W &= B \cup (V \times C \times E) \end{aligned}$$

Die Regeln für die SECDH-Maschine sind analog zu den Regeln für die SECD-Maschine. Zwei Hauptunterschiede gibt es dabei:

- Der Heap aus H gehört nun zum Zustand dazu. Anders als die Umgebung wird er nicht bei der Bildung von Closures „eingepackt“: Stattdessen wird der Heap stets linear von links nach rechts durch Regeln durchgefädelt.
- Zwischenergebnisse nehmen stets den Umweg über den Heap: Immer, wenn ein neues Zwischenergebnis entsteht, wird es bei einer neuen Adresse im Heap abgelegt. Auf dem Stack landen die Adressen der Zwischenergebnisse.

$$\begin{aligned}
& \hookrightarrow \in \mathcal{P}((S \times E \times C \times D \times H) \times (S \times E \times C \times D \times H)) \\
(\underline{s}, e, \underline{b}, \underline{c}, \underline{d}, h) & \hookrightarrow (a\underline{s}, e, \underline{c}, \underline{d}, h[a \mapsto b]) \\
& \text{wobei } a \text{ frisch} \\
(\underline{s}, e, v\underline{c}, \underline{d}, h) & \hookrightarrow (e(v)\underline{s}, e, \underline{c}, \underline{d}, h) \\
(a_n \dots a_1 \underline{s}, e, \text{prim}_F \underline{c}, \underline{d}, h) & \hookrightarrow (a\underline{s}, e, \underline{c}, \underline{d}, h[a \mapsto b]) \\
& \text{wobei } a \text{ frisch, } b_i = h(a_i) \text{ und } F \in \Sigma^{(n)} \text{ und } F_B(b_1, \dots, b_n) = b \\
(a_1 a_0 \underline{s}, e, := \underline{c}, \underline{d}, h) & \hookrightarrow (a\underline{s}, e, \underline{c}, \underline{d}, h[a_0 \mapsto h(a_1)] [a \mapsto \text{void}]) \\
& \text{wobei } a \text{ frisch} \\
(\underline{s}, e, (v, \underline{c}') \underline{c}, \underline{d}, h) & \hookrightarrow (a\underline{s}, e, \underline{c}, \underline{d}, h[a \mapsto (v, \underline{c}', e)]) \\
& \text{wobei } a \text{ frisch} \\
(a_1 a_0 \underline{s}, e, \text{ap} \underline{c}, \underline{d}, h) & \hookrightarrow (\varepsilon, e'[v \mapsto a], \underline{c}', (\underline{s}, e, \underline{c}) \underline{d}, h[a \mapsto h(a_1)]) \\
& \text{wobei } a \text{ frisch und } h(a_0) = (v, \underline{c}', e') \\
(a, e, \varepsilon, (\underline{s}', e', \underline{c}') \underline{d}, h) & \hookrightarrow (a\underline{s}', e', \underline{c}', \underline{d}, h)
\end{aligned}$$

Entsprechend muß die Auswertungsfunktion das Endergebnis im Heap nachschauen:

$$\begin{aligned}
eval_{SECD} & \in \mathcal{L}_{\lambda S} \times Z \\
eval_{SECD}(e) & = \begin{cases} h(a) & \text{falls } (\varepsilon, \emptyset, \llbracket e \rrbracket, \varepsilon, \emptyset) \hookrightarrow^* (a, e, \varepsilon, \varepsilon, h), h(a) \in B \\ \text{proc} & \text{falls } (\varepsilon, \emptyset, \llbracket e \rrbracket, \varepsilon, \emptyset) \hookrightarrow^* (a, e, \varepsilon, \varepsilon, h), h(a) = (v, \underline{c}, e') \end{cases}
\end{aligned}$$

17.8 Implementierung der SECDH-Maschine

Für die Implementierung der SECDH-Maschine werden einige der Prozeduren wiederverwendet, die für die SECD-Maschine programmiert wurden. Zunächst einmal muß – genau wie bei der SECD-Maschine – erst einmal die Übersetzung von Termen in Maschinencode realisiert werden. Zuweisungsterme haben wie in Scheme die folgende Form:

```
(set! v e)
```

Das dazu passende Prädikat ist das folgende:

```
; Prädikat für Zuweisungen
(: assignment? (%a -> boolean))
(define assignment?
  (lambda (t)
    (and (pair? t)
         (equal? 'set! (first t))))))

(define assignment (contract (predicate assignment?)))
```

Mit Hilfe dieser Definition kann die Vertragsdefinition von `term` erweitert werden:

```
(define term
  (contract
    (mixed symbol
      application
      abstraction
      base
      primitive-application
      assignment)))
```

Um zu vermeiden, daß Zuweisungen mit regulären Applikationen verwechselt werden, muß das Prädikat `application?` erweitert werden:

```
(define application?
  (lambda (t)
    (and (pair? t)
         (not (equal? 'set! (first t)))
         (not (equal? 'lambda (first t)))
         (not (primitive? (first t))))))
```

Als nächstes wird die zusätzliche `:=`-Instruktion repräsentiert. Hier sind Daten- und Record-Definition:

```
; Eine Zuweisungs-Instruktion ist ein Wert
; (make-:=)
(define-record-procedures :=
  make-:= :=?
  ())
(: make-:= (-> :=))
```

Die Vertragsdefinition für Maschinen-Instruktionen kann um `:=` erweitert werden:

```
(define instruction
  (contract
    (mixed base
      symbol
      ap
      tailap
      prim
      abs
      :=)))
```

Bei der Übersetzung in Maschinencode kommt in `term->machine-code` ein weiterer Zweig hinzu:

```
; Term in Maschinencode übersetzen
(: term->machine-code (term -> machine-code))
```

```
(define term->machine-code
  (lambda (e)
    (cond
      ...
      ((assignment? e)
       (make-pair (first (rest e))
                  (append (term->machine-code (first (rest (rest e))))
                          (list (make-:=))))))))))
```

Wie bei der SECD-Maschine werden die verschiedenen Mengendefinitionen erst einmal in Daten- und Record-Definitionen übersetzt. Das ist für Stacks, Umgebungen und Speicheradressen ganz einfach:

```
; Ein Stack ist eine Liste aus Adressen.
(define stackh (contract (list address)))

; Eine Umgebung bildet Variablen auf Adressen ab.

; Eine Adresse ist eine ganze Zahl.
(define address (contract natural))
```

Die Änderung in der Definition von Umgebungen bedingt eine Änderung des Vertrags von `make-binding`:

```
(: make-binding (symbol address -> binding))
```

Bei der Repräsentation des Heaps ist wichtig, daß eine Operation zur Beschaffung frischer Adressen eingebaut wird. Aus diesem Grund enthält der Heap zusätzlich zu den Zellen auch noch einen Zähler mit der nächsten frischen Adresse:

```
; Ein Heap ist ein Wert
; (make-heap s n)
; wobei n die nächste freie Adresse ist und s eine Liste
; von Zellen.
(define-record-procedures heap
  make-heap heap?
  (heap-cells heap-next))
(: make-heap ((list cell) natural -> heap))
```

Der leere Heap wird schon einmal vorgefabriziert:

```
(define the-empty-heap (make-heap empty 0))
```

Jede Zelle ordnet einer Adresse einen Wert zu:

```
; Eine Zelle ist ein Wert
; (make-cell a w)
; wobei a eine Adresse und w ein Wert ist
```

```
(define-record-procedures cell
  make-cell cell?
  (cell-address cell-value))
(: make-cell (address value -> cell))
```

Die Prozedur `heap-store`, erweitert den Heap um eine Zelle entsprechend der mathematischen Definition:

```
; Wert im Speicher ablegen
(: heap-store (heap address value -> heap))
(define heap-store
  (lambda (h a w)
    (make-heap (make-pair (make-cell a w)
                          (remove-cell a (heap-cells h)))
              ...)))
```

Die Ellipse steht für die nächste frische Adresse: Wenn die bisherige frische Adresse in `heap-store` belegt wird, so muß eine neue frische Adresse gewählt werden:

```
(define heap-store
  (lambda (h a w)
    (make-heap (make-pair (make-cell a w)
                          (remove-cell a (heap-cells h)))
              (let ((next (heap-next h))
                    (if (= a next)
                        (+ next 1)
                        next))))))
```

Es fehlt noch die Hilfsprozedur `remove-cell`:

```
; Zelle zu einer Adresse entfernen
(: remove-cell (address (list cell) -> (list cell)))
(define remove-cell
  (lambda (a c)
    (cond
      ((empty? c) empty)
      ((pair? c)
       (if (= a (cell-address (first c)))
           (rest c)
           (make-pair (first c)
                      (remove-cell a (rest c))))))))))
```

Als nächstes ist die Operation an der Reihe, die den Wert, der an einer Adresse im Heap gespeichert ist. Die Prozedur `heap-lookup` benutzt eine Hilfsprozedur `cells-lookup`, um in der Liste von Zellen nach der richtigen zu suchen:

```
; den Wert an einer Adresse im Heap nachschauen
```

```
(: heap-lookup (heap address -> value))
(define heap-lookup
  (lambda (h a)
    (cells-lookup (heap-cells h) a)))

; den Wert an einer Adresse in einer Liste von Zellen nachschauen
(: cells-lookup ((list cell) address -> value))
(define cells-lookup
  (lambda (c a)
    (cond
      ((empty? c) (violation "unassigned address"))
      ((pair? c)
       (if (= a (cell-address (first c)))
           (cell-value (first c))
           (cells-lookup (rest c) a))))))
```

Schließlich fehlt noch eine Repräsentation für den void-Wert:

```
; Ein void-Wert ist ein Wert
; (make-void)
(define-record-procedures void
  make-void void?
  ())
(: make-void (-> void))
```

Auch hier wird nur ein void-Wert benötigt, der vorfabriziert wird:

```
(define the-void (make-void))
```

Der Zustand für die SECDH-Maschine wird genau wie bei der SECD-Maschine repräsentiert, ergänzt um die Komponente für den Heap:

```
; Ein SECDH-Zustand ist ein Wert
; (make-secd s e c d h)
; wobei s ein Stack, e eine Umgebung, c Maschinencode,
; d ein Dump und h ein Speicher ist.
(define-record-procedures secdh
  make-secdh secdh?
  (secdh-stack secdh-environment secdh-code secdh-dump secdh-heap))
(: make-secdh (stackh environment machine-code dump heap -> secdh))
```

Die Implementierung der Zustandsübergangsfunktion hat exakt die gleiche Struktur wie die Implementierung der SECD-Maschine und hält sich eng an die mathematische Definition der Regeln:

```
; eine Zustandstransition berechnen
(: secdh-step (secdh -> secdh))
```

```

(define secdh-step
  (lambda (state)
    (let ((stack (secdh-stack state))
          (environment (secdh-environment state))
          (code (secdh-code state))
          (dump (secdh-dump state))
          (heap (secdh-heap state)))
      (cond
        ((pair? code)
         (cond
           ((base? (first code))
            (let ((a (heap-next heap)))
              (make-secdh
               (make-pair a stack)
               environment
               (rest code)
               dump
               (heap-store heap a (first code))))))
           ((symbol? (first code))
            (make-secdh
             (make-pair (lookup-environment environment (first code))
                        stack)
             environment
             (rest code)
             dump
             heap)))
           ((prim? (first code))
            (let ((a (heap-next heap)))
              (make-secdh
               (make-pair a
                          (drop (prim-arity (first code)) stack))
               environment
               (rest code)
               dump
               (heap-store heap a
                           (apply-primitive
                            (prim-operator (first code))
                            (map (lambda (address)
                                   (heap-lookup heap address))
                                 (take-reverse (prim-arity (first code)) stack)))))))))
           ((:=? (first code))
            (let ((a (heap-next heap)))
              (make-secdh
               (make-pair a (rest (rest stack)))
               environment

```

```

      (rest code)
      dump
      (heap-store
       (heap-store heap
                  (first (rest stack))
                  (heap-lookup heap (first stack)))
       a the-void))))
((abs? (first code))
 (let ((a (heap-next heap)))
  (make-secdh
   (make-pair a stack)
   environment
   (rest code)
   dump
   (heap-store heap a
                (make-closure (abs-variable (first code))
                              (abs-code (first code))
                              environment))))))
((ap? (first code))
 (let ((closure (heap-lookup heap (first (rest stack))))
       (a (heap-next heap)))
  (make-secdh empty
              (extend-environment
               (closure-environment closure)
               (closure-variable closure)
               a)
              (closure-code closure)
              (make-pair
               (make-frame (rest (rest stack)) environment (rest code))
               dump)
              (heap-store heap a (heap-lookup heap (first stack))))))
 ((tailap? (first code))
  (let ((closure (heap-lookup heap (first (rest stack))))
        (a (heap-next heap)))
   (make-secdh (rest (rest stack))
              (extend-environment
               (closure-environment closure)
               (closure-variable closure)
               a)
              (closure-code closure)
              dump
              (heap-store heap a
                          (heap-lookup heap (first stack))))))
 ((empty? code)
  (let ((f (first dump)))

```

```
(make-secdh
  (make-pair (first stack)
            (frame-stack f))
  (frame-environment f)
  (frame-code f)
  (rest dump)
  heap))))))
```

Es bleibt die Auswertungsfunktion, die ebenfalls genau wie bei der SECD-Maschine realisiert wird:

```
; aus Term SECDH-Anfangszustand machen
(: inject-secdh (term -> secdh))
(define inject-secdh
  (lambda (e)
    (make-secdh empty
                the-empty-environment
                (term->machine-code e)
                empty
                the-empty-heap)))

; bis zum Ende Zustandsübergänge berechnen
(: secdh-step* (secdh -> secdh))
(define secdh-step*
  (lambda (state)
    (if (and (empty? (secdh-code state))
            (empty? (secdh-dump state)))
        state
        (secdh-step* (secdh-step state))))))

; Evaluationsfunktion zur SECD-Maschine berechnen
(: eval-secdh (term -> (mixed value (one-of 'function))))
(define eval-secdh
  (lambda (e)
    (let ((final (secdh-step* (inject-secdh e))))
      (let ((val (heap-lookup (secdh-heap final)
                             (first (secdh-stack final)))))
        (if (base? val)
            val
            'proc))))))
```

Übungsaufgaben

Aufgabe 17.1 Übersetzen Sie folgende Lambda-Terme in die Zwischenrepräsentation der SECD-Maschine:

1. $(\lambda xy. (+ xy)) (* 5 6) 23$
2. $(\lambda x. (!x)) (\lambda xy. (&& xy)) ((\lambda xy. (> xy)) 23 42) true$
3. $(\lambda xy. yxx) (\lambda z. z) (\lambda yz. (yy) (yz))$

Dabei steht ! für das boolesche not und && für das boolesche and.

Aufgabe 17.2 Betrachten Sie folgendes SECD-Programm:

$$(f, (x, (y, f x a p y a p))) (a, (b, a b \text{prim}_+)) \text{ap } 23 \text{ ap } 42 \text{ ap}$$

1. Übersetzen Sie das SECD-Programm in den entsprechenden $\mathcal{L}_{\lambda A}$ -Term.
2. Werten Sie das SECD-Programm aus und geben Sie die einzelnen Auswertungsschritte an!

Aufgabe 17.3 Erweitern Sie die Implementierung der SECD-Maschine um korrekte Behandlung der Endrekursion! Erweitern Sie dazu zunächst die Datendefinition für Maschinencode. Implementieren Sie dann die Übersetzung von λ -Termen für die endrekursive SECD-Maschine. Erweitern Sie schließlich die Zustandsübergangsfunktion um einen Fall für die `tailap`-Instruktion.

Aufgabe 17.4 Die um Endrekursion erweiterte SECD-Maschine führt eine neue Maschinencode-Instruktion `tailap` ein. Dies ist aber nicht unbedingt nötig. Formulieren Sie die Zustandsübergangsregeln der SECD-Maschine mit Endrekursion so um, daß die Funktionalität, also insbesondere die richtige Behandlung endrekursiver Applikationen, auch ohne das zusätzliche Schlüsselwort `tailap` erhalten bleibt.

Aufgabe 17.5 Zeigen Sie in der um Endrekursion erweiterten SECD-Maschine, daß `tailap` immer am Ende steht, also tatsächlich keinen Kontext besitzt.

Aufgabe 17.6 Erweitern Sie die SECD-Maschine um Primitive anderer Stelligkeiten, z.B. `abs` oder `odd?`.

Aufgabe 17.7 Ändern Sie die Implementierung der SECDH-Maschine dahingehend, daß sie Endrekursion korrekt behandelt.

Aufgabe 17.8 Abstrahieren Sie über `remove-environment-binding` und `remove-cell`.

Aufgabe 17.9 Erweitern Sie den angewandten λ -Kalkül um Abstraktionen und Applikationen mit mehr als einem Parameter. Erweitern Sie die SECD-Maschine und ihre Implementierung entsprechend.

Aufgabe 17.10 Erweitern Sie den angewandten λ -Kalkül um binäre Verzweigungen analog zu `if`. Erweitern Sie entsprechend die SECD-Maschine und ihre Implementierung.

Aufgabe 17.11 `begin` läßt sich im angewandten λ -Kalkül als syntaktischer Zucker auffassen: Wie müßten `begin`-Ausdrücke in die Sprache des Kalküls übersetzt werden?

Aufgabe 17.12 Anstatt Umgebungen durch Listen von Bindungen zu repräsentieren, ist es auch möglich, Prozeduren zu verwenden, so daß `lookup-environment` folgendermaßen aussieht:

```
(define lookup-environment
  (lambda (e v)
    (e v)))
```

Ergänzen Sie eine passende Definition für `extend-environment`.

Aufgabe 17.13 Auf den ersten Blick erscheint es etwas aufwendig, jedesmal bei der Auswertung einer Abstraktion die gesamte Umgebung in die Closure einzupacken. Was würde sich ändern, wenn dieser Schritt weggelassen würde, Closures also nur Variable und Maschinencode für den Rumpf enthalten würden? Formulieren Sie die entsprechenden Regeln für die SECD-Maschine und ändern Sie die Implementierung entsprechend. Funktioniert die SECD-Maschine nach der Änderung noch korrekt?