Eine Einführung in Monaden

Chang Liu
liuc@mathematik.uni-marburg.de
Betreuer: Prof. Dr. H. Peter Gumm

18. Juli 2005

Abstrakt: *Monade* ist ein mathematischer Begriff in der Kategorientheorie, der in den 60er Jahre erfunden wurde. List-Comprehensions ist ein wichtiges Konzept in funktionalen Sprachen, die mit Liste zu tun. In [1] zeigt man, wie man aus List-Comprehensions eine Monade generalisiert, damit man in reiner funktionalen Sprachen viele Nebenwirkungen realisieren kann, wie z.B. Zustandsverarbeitung, Exceptionsbehandlung, Continuations, usw..

Inhaltsverzeichnis

1 Einführung

In einer funktionalen Programmiersprache werden Berechnungen als Auswertung mathematischer Funktionen verstanden, deshalb ist sie für mathematische Beweise gut geeignet. Es gibt unterschiedliche funktionale Sprachen, allerdings wird dazu oft nur bestimmte Type von Sprachen benutzt, die sogenannten reinen funktionalen Sprachen und unreinen funktionalen Sprachen.

Unreine, strikte funktionale Sprachen wie Standard ML, Scheme, unterstützen viele Nebenwirkungen, wie z.B. Zustandsverarbeitung, Exceptionsbehandlung, Continuations usw.. Reine funktionale Sprachen wie Haskell, Miranda haben aber keine solche Nebenwirkungen, weil bei ihnen alle Berechnungen nur durch Funktionsanwendung durchgeführt werden.

Man hat lange nach möglichen Wegen gesucht, um die Vorzüge von unreinen Sprachen mit der leichten Beweisbarkeit in reinen Sprachen kombinieren zu können. Das Konzept *Monaden* in der Kategorientheorie findet man gut geeignet, wenn man es auf die funktionale Programmierung anwendet. Im Folgenden soll die Definition von Monaden gegeben werden, dazu werden zwei wichtige Monaden, die Identitätsmonade und die State-Transformersmonade, sowie ein Interpreter-Beispiel gezeigt.

2 Hintergrund

2.1 Listen

Listen stellen die wichtigste Datenstruktur in Haskell dar.

$$data[a] = [\,] \mid a : [a]$$

Sie werden benutzt, um eine beliebige Zahl von Werten gleichen Typs zusammenzufassen. Wie z.B. $[1, 2, 3] :: [Int]$, $['a', 'b', 'c'] :: [Char]$.

Eine der wichtigsten Funktionen in Haskell ist die *map* Funktion. Sie nimmt als Argumente eine Funktion und eine Liste und liefert als Ergebnis die Liste, die durch Anwenden der Funktion auf alle Elemente der Argumentliste entsteht:

$$\begin{array}{lcl} map & :: & (a \to b) \to [a] \to [b] \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

Z.B. ist $map\ (3*)[1,2,3] = [3,6,9]$. Und für die *map* Funktion gibt es zwei wichtige Eigenschaften:

$$\begin{array}{lll} (i) & map\ id & = \quad id \\ (ii) & map\ (g \circ f) & = \quad map\ g \circ map\ f \end{array}$$

Hier ist id die Identitätsfunktion und $g \circ f$ Komposition von Funktion g und f, $(g \circ f)x = g(fx)$.

Dann verallgemeinern wir die Listen aus Konstruktionsprinzip M. $M\ x$ steht für den Datentyp von Listen mit Elementen von Typ x. Wie z.B. $[1,2,3] :: M\ Int$, $['a','b','c'] :: M\ Char$. (In Haskell sind Typvariablen in kleinen Buchstaben geschrieben, wie z.B x und y, und Typkonstruktoren sind in großen Buchstaben geschrieben, z.B. M.) Die *map* Funktion wird jetzt definiert als Funktion von Typ:

$$map \quad :: \quad (x \to y) \to (M\ x \to M\ y)$$

und mit Eigenschaften (i) und (ii). Z.B sei $code :: Char \to Int$ eine Funktion, die den ASCII-Wert von einem Buchstabe rechnet, dann ist $map\ code\ ['a','b','c'] = [97,98,99]$.

2.2 Funktoren

In Kategorientheorie versteht man unter einem *Funktor* $M : K \to L$ eine Zuordnung, durch die zu jedem Objekt $x \in K$ ein Objekt $M\ x \in L$ und zu jedem Morphismus $f : x \to y$ (wobei x, y Objekt von K sind) ein Morphismus $M_f : M\ x \to M\ y$ gegeben ist, und darüber hinaus Folgendes erfüllt:

$$\begin{array}{lll} M_{id_x} & = & id_{M\ x} \\ M_{g \circ f} & = & M_g \circ M_f \end{array}$$

Notation 2.1. *In funktionalen Sprachen schreibt man oft $M_f = map\ f$, oder $map^M\ f$*

Beispiel 2.1. *Der List-Funktor $M = [\ ]$.*

Die Eigenschaften des List-Funktors ist wie im Abschnitt 2.1 gezeigt. Für den List-Funktor gibt es noch zusätzliche Operationen:

$$\begin{array}{lll} unit & :: & x \to [x] \\ join & :: & [[x]] \to [x] \end{array}$$

$$\begin{array}{ccccc} x & \xrightarrow{unit} & [x] & \xleftarrow{join} & [[x]] \\ {\scriptstyle f}\downarrow & & {\scriptstyle map\ f}\downarrow & & {\scriptstyle map(map\ f)}\downarrow \\ y & \xrightarrow{unit} & [y] & \xleftarrow{join} & [[y]] \end{array}$$

Abbildung 1: Das kommutative Diagramm von List-Funktor

Die Funktion unit konvertiert einen Wert in einer einelementigen List, und die Funktion join ist wie die Funktion concat $::\ [[a]] \to [a]$ *in Haskell definiert, die eine List von Listen in einer List konvertiert. Wie z.B.* $unit\ 3 = [3]$, $join\ [[1],[2,3]] = [1,2,3]$. *Für die zwei Funktionen gibt es zwei wichtige Regeln:*

$$\begin{array}{lll} (iii) & map\ f \circ unit & = \quad unit \circ f \\ (iv) & map\ f \circ join & = \quad join \circ map(map\ f) \end{array}$$

Dann bekommt man das kommutative Diagramm mit natürlichen Transformationen wie in Abbildung 1.

3 Monaden

3.1 Monaden und Comprehensions

3.1.1 Monaden

Eine *Monade* ist ein Funktor M, d.h. er erhält Eigenschaften (i) und (ii), mit natürlichen Transformationen:

$$\begin{array}{lll} unit & :: & x \to M\ x \\ join & :: & M\ M\ x \to M\ x, \end{array}$$

d.h. sie erfüllen Regeln (iii) und (iv), wie in Abbildung 2(a) gezeigt ist. Zusätzlich sollen noch die *unit*-Gleichung und die Assoziativität gelten:

$$\begin{array}{lll} (I) & join \cdot unit & = \quad id \\ (II) & join \cdot map\ unit & = \quad id \\ (III) & join \cdot join & = \quad join \cdot map\ join \end{array}$$

Diese Regeln sind diagrammatisch in Abbildung 2(b),(c) ausgedrückt.

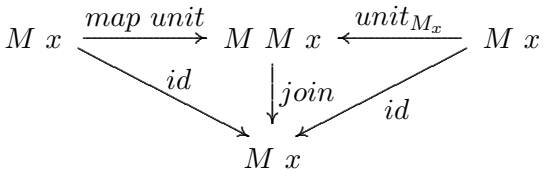
Notation 3.1. $map^M, unit^M, join^M$ *steht für die Monade* M.

$$\begin{array}{ccccc} x & \xrightarrow{unit} & M\ x & \xleftarrow{join} & M(M\ x) \\ {\scriptstyle f}\downarrow & & {\scriptstyle map\ f}\downarrow & & {\scriptstyle map(map\ f)}\downarrow \\ y & \xrightarrow{unit} & M\ y & \xleftarrow{join} & M(M\ y) \end{array}$$

(a) Die allgemeine natürliche Transformationen (*unit* und *join*).

$$\begin{array}{ccc} MMM\ x & \xrightarrow{map\ join} & MM\ x \\ {\scriptstyle join_{Mx}}\downarrow & & {\scriptstyle join}\downarrow \\ MM\ x & \xrightarrow{join} & M\ y \end{array}$$

(b) Die Assoziativität.



(c) Die *unit*-Gleichung.

Abbildung 2: Die kommutative Diagramme von Monaden.

3.1.2 Comprehensions

In Haskell hat eine Listabstraktion [2] (list comprehensions) die Form:

$$[expr \mid q_1, \ldots, q_n]$$

wobei die $q_i (1 \leq i \leq n)$ von der folgenden Form sein können:

- *Generatoren* der Form $pat <- listexp$, wobei das Muster *pat* den Typ t hat und *listexp* eine Liste von Typ $[t]$ definiert.
- *Bedingungen*, d.h. beliebige Ausdrücke vom Typ *Bool*.
- *lokale Bedingungen*, d.h. durch *let* eingeleitete Definitionen von Bezeichnern, die in nachfolgenden Generatoren und Bedingungen benutzt werden können.

Beispiel 3.1. $[(x, y) | x \leftarrow [1, 2], y \leftarrow [3, 4]] = [(1, 3), (1, 4), (2, 3), (2, 4)]$.

Wir verallgemeinern die List-Comprehensions zu allgemeinen Monaden-Comprehensions. Eine *Comprehension* hat die Form:

$$[t \mid q]$$

wobei t ein Term ist und q ein Qualifier ist. Ein Qualifier ist:

- *Leer*, Λ

- ein *Generator* $x \leftarrow u$, wobei x eine Variable ist und u ein Ausdruck mit Wert in $M\ x$ ist.
- *Komposition* von Qualifiern (p, q)

Die Comprehension ist definiert durch die folgenden Regeln:

$$\begin{array}{lll} (1) & [t\ |\Lambda] & = \ unit\ t \\ (2) & [t\ |x \leftarrow u] & = \ map\ (\lambda x \rightarrow t)\ u \\ (3) & [t\ |\ (p,q)] & = \ join[\ [\ t\ |\ q\ ]\ |\ p\ ] \end{array}$$

Beispiel 3.2. $[(x,y)|x \leftarrow [1,2], y \leftarrow [3,4]]$

$$\begin{aligned} [(x,y)|x \leftarrow [1,2], y \leftarrow [3,4]] &=^{(3)} join[[(x,y)|y \leftarrow [3,4]]|x \leftarrow [1,2]] \\ &=^{(2)} join[map(\lambda y \leftarrow (x,y))[3,4]|x \leftarrow [1,2]] \\ &=^{(2)} join[map(\lambda x \leftarrow map(\lambda y \leftarrow (x,y))[3,4])[1,2]] \\ &= join[map(\lambda x \leftarrow [(x,3),(x,4)])[1,2]] \\ &= join[[(1,3),(1,4)],[(2,3),(2,4)]] \\ &= [(1,3),(1,4),(2,3),(2,4)] \end{aligned}$$

Aus Regeln $(i) - (iv)$ und $(1) - (3)$ können wir folgern:

$$\begin{array}{lll} (4) & [f\ t\ |\ q] & = \ map\ f\ [t\ |\ q] \\ (5) & [x\ |x \leftarrow u] & = \ u \\ (6) & [t\ |p, x \leftarrow [u|q], r] & = \ [t^u_x|p, q, r^u_x] \end{array}$$

wobei in (4) die Funktion f darf keine Vorkommen von in q gebundenen Variablen haben, und in (6) der Term t^u_x steht für den Term t, in dem jede freie Vorkommen von der Variable x durch den Term u ersetzt werden.

Beweis 3.1.

$$(4) \quad [f\ t\ |\ q] \quad = \quad map\ f\ [t\ |\ q]$$

Beweis durch strukturelle Induktion über q
I.A: $q = \Lambda$

$$\begin{aligned} [f\ t\ |q] &= [f\ t\ |\Lambda] \\ &\Rightarrow^{(1)} unit\ f\ t \\ &\Rightarrow^{(iii)} map\ f \cdot unit\ t \\ &\Rightarrow^{(1)} map\ f\ [t|\Lambda] = map\ f[t|q] \end{aligned}$$

$q = x \leftarrow u$

$$\begin{aligned}
[ft|q] &= [ft|x \leftarrow u] \\
&\Rightarrow^{(2)} map(\lambda x \rightarrow f\ t)\ u \\
&= map\ (f \circ (\lambda x \rightarrow t))\ u \\
&\Rightarrow^{(ii)} map\ f \circ map(\lambda x \rightarrow t)\ u \\
&\Rightarrow^{(2)} map\ f\ [t|x \leftarrow u] \\
&= map\ f\ [t|q]
\end{aligned}$$

I.S: $q = (p, q)$

$$\begin{aligned}
[f\ t|\ q] &= [f\ t|\ (p, q)] \\
&\Rightarrow^{(3)} join\ [[f\ t|\ q]|\ p] \\
&\Rightarrow^{\mathbf{I.V}} join\ [map\ f[t|\ q]|\ p] \\
&\Rightarrow^{\mathbf{I.V}} join\ map\ (map\ f)[[t|\ q]|\ p] \\
&\Rightarrow^{(iv)} map\ f \circ join\ [[t|\ q]|\ p] \\
&\Rightarrow^{(3)} map\ f\ [t|\ (p, q)]
\end{aligned}$$

□

Beweisskizze für (5) **und** (6)**:** (5) ist eine direkte Folger von Regeln (i) und (2). Für (6) benutzen wir wieder strukturelle Induktion über q mit Hilfe von Regeln $(1) - (4)$.

3.1.3 Umwandeln von Monaden und Comprehensions

Aus einer Comprehension können wir *unit*, *map* und *join* gewinnen:

$(1')$	$unit\ x$	$=$	$[x]$
$(2')$	$map\ f\ \bar{x}$	$=$	$[f\ x|x \leftarrow \bar{x}]$
$(3')$	$join\ \bar{\bar{x}}$	$=$	$[x|\bar{x} \leftarrow \bar{\bar{x}}, x \leftarrow \bar{x}]$

wobei x Typ von x, $\bar{x}$ Typ von $M\ x$, $\bar{\bar{x}}$ Typ von $M\ M\ x$ ist.

Beispiel 3.3.

$$[x\ |x \leftarrow [1, 2]] = map\ id\ [1, 2]$$

Wir können nicht nur aus Comprehensions Monaden folgern, sondern auch aus Monaden Comprehensions. Aber zuerst müssen wir eine Comprehensionsstruktur definieren. Eine Comprehensionsstruktur ist eine Comprehension mit Regeln (5), (6) und

$$\begin{array}{lll} (I') & [t|\Lambda, q] = & [t|q] \\ (II') & [t|q, \Lambda] = & [t|q] \\ (III') & [t|(p,q), r] = & [t|p, (q,r)] \end{array}$$

Dann mit Regeln (1) – (3) können wir die Monaden in Comprehensionsstruktur umwandeln.

3.2 Die Identitätsmonade

Die Identitätsmonade ist eine triviale Monade mit Eigenschaften:

$$\begin{array}{lll} type\ Id\ x & = & x \\ map^{Id} f x & = & f\ x \\ unit^{Id} x & = & x \\ join^{Id} x & = & x \end{array}$$

Die Funktionen $map^{Id}, unit^{Id}, join^{Id}$ sind alle Identitätsfunktionen. Eine Comprehension in der Identitätsmonade entspricht einem "Let" Term:

$$[t|x \leftarrow u]^{Id} = ((\lambda x \rightarrow t)u) = (let\ x = u\ in\ t)$$

Ähnlich ist

$$[t|x \leftarrow u, y \leftarrow v]^{Id} = (let\ x = u\ in\ (let\ y = v\ in\ t))$$

.

4 Zustandsverarbeitung

Zustandsverarbeitung spielt eine wichtige Rolle in Programmiersprachen. In unreiner funktionalen Sprache ist sie direkt möglich, aber in einer reiner funktionalen Sprache muss man durch einen zusätzlichen Wert den aktuellen Zustand repräsentieren. In diesem Abschnitt werden wir zeigen, wie man mit Hilfe von State-Transformersmonade und ihrer entsprechenden Comprehension das Programm strukturieren.

$$\begin{array}{ccccc} x & \xrightarrow{unit} & (x,S)^S & \xleftarrow{join} & ((x,S)^S,S)^S \\ {\scriptstyle f}\downarrow & & {\scriptstyle map\ f}\downarrow & & {\scriptstyle map(map\ f)}\downarrow \\ y & \xrightarrow{unit} & (y,S)^S & \xleftarrow{join} & ((y,S)^S,S)^S \end{array}$$

Abbildung 3: Das kommutative Diagramm von State-Transformers.

4.1 State-Transformers

Sei S Typ von Zustand, die State-Transformersmonade ST ist definiert durch:

$$\begin{array}{lcl} type\ STx & = & S \to (x,S) \\ map^{ST} f \bar{x} & = & \lambda s \to [(fx,s')|(x,s') \leftarrow \bar{x}s]^{Id} \\ unit^{ST} x & = & \lambda s \to (x,s) \\ join^{ST} \bar{\bar{x}} & = & \lambda s \to [(x,s'')|(\bar{x},s') \leftarrow \bar{\bar{x}}s, (x,s'') \leftarrow \bar{x}s]^{Id} \end{array}$$

Das entsprechende kommutative Diagramm ist wie in Abbildung 3 gezeigt.

Ein State-Transformer von Typ x liest einen Zustand ein und gibt den Wert von x und einen neuen Zustand aus. Die Funktion $unit$ nimmt den Wert x in dem State-Transformer $\lambda s \to (x,s)$ ein, dann gibt x aus und lässt den Zustand unverändert. Die entsprechende Comprehension ist:

$$[(x,y)|x \leftarrow \bar{x}, y \leftarrow \bar{y}]^{ST} = \lambda s \to [((x,y),s'')|(x,s') \leftarrow \bar{x}s, (y,s'') \leftarrow \bar{y}s']^{Id}$$

Sie wendet eine State-Transformer $\bar{x}$ auf einem Zustand x an, erzeugt den Wert x und einen neuen Zustand s'; dann wendet sie eine andere State-Transformer $\bar{y}$ auf dem Zustand s' an, erzeugt den Wert y und einen neueren Zustand s'', am Ende gibt sie ein Tupel $((x,y),s'')$ als Endergebnis aus, wobei (x,y) Wert und s'' Endzustand ist.

Drei wichtigste Funktionen dieser Monade sind:

$$\begin{array}{lcl} fetch & :: & ST\ S \\ fetch & = & \lambda s \to (s,s) \\ assign & :: & S \to ST() \\ assign\ s' & = & \lambda s \to ((),s') \\ init & :: & S \to ST\ x \to x \\ init\ s\ \bar{x} & = & [x\ |(x,s') \leftarrow \bar{x}s]^{Id} \end{array}$$

Die $fetch$ Funktion holt den aktuellen Zustand ab und lässt den Zustand unverändert; die $assign$ Funktion liefert einen neuen Zustand, ohne den alten Zustand zu betrachten. Hier ist () der Datentyp, der nur den Wert () enthält. Die $init$ Funktion wendet den State-Transformer $\bar{x}$ auf einem gegebenem Zustand s an, und gibt nur den Wert aus, der durch die State-Transformer gerechnet wird, ohne den Endzustand zu beobachten.

4.2 Array update

Sei Arr der Typ von Array, sein Index von Typ Ix und sein Element(Wert) Typ von Val ist. Die wichtigen Operationen von diesem Typ sind:

$$\begin{array}{lll} newarray & :: & Val \to Arr \\ index & :: & Ix \to Arr \to Val \\ update & :: & Ix \to Val \to Arr \to Arr \end{array}$$

Hier gibt $newarray\ v$ einen Array, der an jeder Position den Wert v gespeichert hat, zurück, und $index\ i\ a$ gibt den Wert $a[i]$ zurück, und $update\ i\ v\ a$ gibt eine neue Array zurück, in der das i-te Element der Wert v ist, und die anderen identisch zu denen in a sind.

Wenn in dem State-Transformer der Zustand von Typ $S = Arr$ ist, dann ist

$$type\ ST\ x = Arr \to (x, Arr)$$

Die $fetch$ und $assign$ Operationen jetzt können auf einen Array angewendet werden.

$$\begin{array}{lll} fetch & :: & Ix \to ST\ Val \\ fetch\ i & = & \lambda a \to [(v,a)|v \leftarrow index\ i\ a]^{Str} \\ assign & :: & Ix \to Val \to ST() \\ assign\ i\ v & = & \lambda a \to ((), update\ i\ v\ a) \\ init & :: & Val \to ST\ x \to x \\ init\ v\ \bar{x} & = & [x|(x,a) \leftarrow \bar{x}(newarray\ v)]^{Id} \end{array}$$

Die Str-Comprehension hier bedeutet: falls die $index\ i\ a$ definiert ist, ist $v = index\ i\ a$, anderenfalls ist $v = \perp(undef.)$. Dies garantiert, dass die Operation $fetch$ immer definiert ist.

4.3 Beispiel: Interpreter

Nehmen wir an, dass wir einen Interpreter für eine einfache imperative Sprache bauen wollen. Die Speicherweise der Sprache wird durch einen Zustand von Arr modelliert, so dass wir Ix als den Typ von Variablennamen betrachten können, und Val als den Typ von Werten, die in Variablen gespeichert sind. Die abstrakte Syntax dieser Sprache ist wie folgt definiert:

$$\begin{array}{lcl} data\ Exp & = & Var\ Ix \mid Const\ Val \mid Plus\ Exp\ Exp \\ data\ Com & = & Asgn\ Ix\ Exp \mid Seq\ Com\ Com \mid If\ Exp\ Com\ Com \\ data\ Prog & = & Prog\ Com\ Exp \end{array}$$

Eine Expression ist eine Variable, eine Konstante, oder die Summe von zwei Expressions; ein Command ist eine Zuweisung, eine Sequenz von zwei Commands, oder eine If-Anweisung; und ein Programm enthält ein Command mit einer Expression danach.

Eine Version des Interpreters in einer reinen funktionalen Sprache ist wie in Abbildung 4 gezeigt. Der Interpreter kann als eine denotationelle Semantik für die Sprache ausgesehen werden, mit drei Semantik-Funktionen *exp*, *com* und *prog*. Die Semantik einer Expression liest einen Array ein und gibt einen Wert aus; die Semantik eines Commands liest einen Array ein und gibt einen neuen Array aus; und die Semantik eines Programms ist ein Wert. Ein Programm enthält ein Command und eine Expression danach, der Wert davon ist durch das Command bestimmt, das auf einem Array mit allen Entitäten 0 angewendet wird. Aus der Expression bekommt man das Ergebnis.

Der gleiche Interpreter kann mit State-Transformers geschrieben werden, wie in Abbildung 5 gezeigt ist. Die Semantik-Funktionen haben jetzt andere Typen. Die Semantik einer Expression ist von dem Zustand abhängig und gibt einen Wert zurück; die Semantik eines Commands transformiert nur den Zustand; und die Semantik eines Programm ist wieder ein Wert.

Im Vergleich zu dem Interpreter mit reiner funktionalen Sprache ist der Interpreter mit State-Transformers gut strukturiert und besser lesbar.

5 Zusammenfassung

Im Vergleich zu unreiner funktionalen Sprache haben reine funktionale Sprache keinen Seiteneffekt, deshalb sind ihre Ausdrucksmöglichkeiten begrenzt. Um diese Beschränkung zu verbessern, benutzt man das Konzept der “Monade“

$$
\begin{array}{lcl}
exp & :: & Exp \to Arr \to Val \\
exp(Var\ i)\ a & = & index\ i\ a \\
exp(Const\ v)\ a & = & v \\
exp(Plus\ e_1\ e_2)\ a & = & exp\ e_1\ a\ + exp\ e_2\ a \\
com & :: & Com \to Arr \to Arr \\
com(Asgn\ i\ e)\ a & = & update\ i\ (exp\ e\ a)\ a \\
com(Seq\ c_1\ c_2)\ a & = & com\ c_2\ (com\ c_1\ a) \\
com(If\ e\ c_1\ c_2)\ a & = & if\ exp\ e\ a = 0\ then\ com\ c_1\ a\ else\ com\ c_2\ a \\
prog & :: & Prog \to Val \\
prog(Prog\ c\ e) & = & exp\ e\ (com\ c\ (newarray\ 0))
\end{array}
$$

Abbildung 4: Der Interpreter in einer reinen funktionalen Sprache.

$$
\begin{array}{lcl}
exp & :: & Exp \to ST\ Val \\
exp(Var\ i) & = & [v|\ v \leftarrow fetch\ i]^{ST} \\
exp(Const\ v) & = & [v]^{ST} \\
exp(Plus\ e_1\ e_2) & = & [v_1 + v_2 | v_1 \leftarrow exp\ e_1, v_2 \leftarrow exp\ e_2]^{ST} \\
com & :: & Com \to ST() \\
com(Asgn\ i\ e) & = & [()|v \leftarrow exp\ e, () \leftarrow assign\ i\ v]^{ST} \\
com(Seq\ c_1\ c_2) & = & [()|() \leftarrow com\ c_1, () \leftarrow comc_2]^{ST} \\
com(If\ e\ c_1\ c_2) & = & [()|v \leftarrow exp\ e, () \leftarrow if\ v = 0\ then\ com\ c_1\ else\ com\ c_2]^{ST} \\
prog & :: & Prog \to Val \\
prog(Prog\ c\ e) & = & init\ 0[() \leftarrow com\ c, v \leftarrow exp\ e]^{ST}
\end{array}
$$

Abbildung 5: Der Interpreter mit State-Transformers.

aus der Kategorientheorie. Sie bietet die Möglichkeit, Programme mit Seiteneffekten zu strukturieren. Die Identitätsmonade ist eine triviale Monade, und die State-Transformer ist eine der wichtigen Monaden für die Modellierung von Zustandsveränderungen.

Literatur

[1] Phil Wadler: Comprehending Monads, Mathematical Structures in Computer Science, Vol. 2, pp 461-493, Cambridge Univ. Press 1992.

[2] Prof. Dr. Rita Loogen: Skript zur Vorlesung Praktische Informatik III: Deklarative Programmierung. Wintersemester 2003/2004.