Funktionale Programmierung

Monaden

Andreas Abel, Hans-Wolfgang Loidl

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

30. April 2009

Monaden

- Eine *Monade* ist ein Programmier-Schema für sequentielle Berechnungen.
- In Haskell werden Berechnungen mit Effekten (wie Zustand, Ein-/Ausgabe, Ausnahmen) mittels Monaden simuliert.
- Monaden sind *keine* Spracherweiterung, man könnte alles rein funktional zu Fuß programmieren.
- Jedoch gibt es syntaktische Unterstützung für Monaden, jedoch auch die selbst-programmierten Monaden.
- Sie helfen lediglich bei der Strukturierung von Code (Modularität, Lesbarkeit).
- Die Schnittstelle zur *Außenwelt* läuft über die *IO*-Monade.

Parser als effektvolle Berechnung

1. Zustand: Die Eingabe wird um den geparsten Teil verkürzt.

   $$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to (a, \mathit{String})$$

   In der rein funktionalen Welt wird der Zustand *durchgeschleift*, ist also Eingabe und Ausgabe des Parsers.

2. Nicht-Determinismus: Mehrere Ergebnisse sind möglich, realisiert als Liste.

   $$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

   Wir verwenden nur zwei mögliche Ergebnisse: Die leere Liste $[]$ ist *Ausnahme*, die einelementige $[(a, \mathit{inp}')]$ ist reguläres Ergebnis.

Parser als sequentielle Berechnung

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

Zwei Parser werden mit *bindP* in Sequenz geschaltet.

$$\begin{aligned}
&\mathit{bindP} \quad :: \mathit{Parser}\ a \to (a \to \mathit{Parser}\ b) \to \mathit{Parser}\ b \\
&\mathit{bindP}\ p\ k = \lambda \mathit{inp} \to \textbf{case}\ (p\ \mathit{inp})\ \textbf{of} \\
&\qquad\qquad [\,] \to [\,] \\
&\qquad\qquad [(a, \mathit{inp}')] \to (k\ a)\ \mathit{inp}'
\end{aligned}$$

returnP ist völlig effekt*neutral*, es kann keine Aufnahme werfen und gibt die Eingabe ungesehen weiter.

$$\begin{aligned}
&\mathit{returnP} \quad :: a \to \mathit{Parser}\ a \\
&\mathit{returnP}\ a = \lambda \mathit{inp} \to [(a, \mathit{inp})]
\end{aligned}$$

Gesetze der Sequenz

1. Links-Eins:

$$(returnP\ a)\ `bindP`\ (\lambda a' \rightarrow k\ a') = k\ a$$

2. Rechts-Eins:

$$p\ `bindP`\ (\lambda a \rightarrow returnP\ a) = p$$

3. Assoziativität:

$$(p\ `bindP`\ \lambda a \rightarrow q\ a)\ `bindP`\ \lambda b \rightarrow k\ b =$$
$$p\ `bindP`\ \lambda a \rightarrow (q\ a\ `bindP`\ \lambda b \rightarrow k\ b)$$

Monaden-Definition

Eine *Monade* ist ein Tripel $(M, return, (\gg\!=))$ (sprich "*bind*" für $\gg\!=$) mit folgender Typisierung:

$$\textbf{type } M\ a$$
$$return :: a \to M\ a$$
$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$

Folgende Gesetze müssen gelten:

$$\begin{array}{lll}
return\ a \gg\!= k & = k\ a & \text{-- Links-Eins} \\
m \gg\!= return & = m & \text{-- Rechts-Eins} \\
(m \gg\!= \lambda a \to k\ a) \gg\!= k' = & & \text{-- Assoziativität} \\
m \gg\!= \lambda a \to (k\ a \gg\!= k') & &
\end{array}$$

Monade ist abgeleitet von *Monoid*. In der Kategorientheorie bezeichnet man $(M, return, (\gg\!=))$ als *Kleisli-Tripel*.

Fehler-Monade

Eine Berechnung liefert entweder ein Ergebnis a oder wirft die Ausnahmne s.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Except}\ a &&= \mathit{Fail}\ \mathit{String}\\
&&&\mid \mathit{Ok}\ \ a\\
&\mathit{returnE} &&:: a \to \mathit{Except}\ a\\
&\mathit{returnE}\ a &&= \mathit{Ok}\ a\\
&\mathit{bindE} &&:: \mathit{Except}\ a \to (a \to \mathit{Except}\ b) \to \mathit{Except}\ b\\
&\mathit{bindE}\ (\mathit{Fail}\ s)\ k &&= \mathit{Fail}\ s\\
&\mathit{bindE}\ (\mathit{Ok}\ a)\ \ k &&= k\ a
\end{aligned}
$$

Variante mit parametrisiertem Fehlertyp:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Except}\ s\ a = \mathit{Fail}\ s\\
&\qquad\qquad\qquad\mid \mathit{Ok}\ \ a
\end{aligned}
$$

In der Praxis verwendet man hierfür *Either*.

Beispiel: Division

```
divideE       :: Except Int → Except Int → Except Int
divideE m n = m `bindE` (λx →
                n `bindE` (λy →
                  if y ≡ 0 then Fail "division by zero"
                           else returnE (x `div` y)))
```

In Haskell kann man das viel leserlicher schreiben.

```
divideE       :: Except Int → Except Int → Except Int
divideE m n = do x ← m
                 y ← n
                 if y ≡ 0 then fail "division by zero"
                          else return (x `div` y)
```

Sieht schon fast aus wie ein imperatives Programm!

do-Notation

Um die **do**-Notation benutzen zu können, muss man Haskell mitteilen, dass *Except* eine Monade ist:

instance *Monad Except* **where**
$$\begin{aligned} return &= returnE \\ (\gg=) &= bindE \\ fail &= Fail \end{aligned}$$

1. Setzt das überladene *return*.
2. Setzt das überladene $\gg=$ und installiert die **do**-Notation.
3. (optional) Setzt die Null der Monade auf *Fail*. Standardmäßig ist sie *error*. Für die Null gilt:

$$fail\ s \gg= k = fail\ s$$

Weiteres Aufhübschen

Haskell definiert:

$$
\begin{aligned}
(\$) \quad &:: (a \to b) \to a \to b \\
f\ \$\ a &= f\ a
\end{aligned}
$$

Und im Modul *Control.Monad*:

$$
\begin{aligned}
\mathit{when} \quad &:: \mathit{Monad}\ m \Rightarrow \mathit{Bool} \to m\ () \to m\ () \\
\mathit{when}\ \mathit{True}\ \ m &= m \\
\mathit{when}\ \mathit{False}\ m &= \mathit{return}\ ()
\end{aligned}
$$

Damit reduzieren wir Schachtelung und Klammern:

$$
\begin{aligned}
\mathit{divideE} \quad &:: \mathit{Except}\ \mathit{Int} \to \mathit{Except}\ \mathit{Int} \to \mathit{Except}\ \mathit{Int} \\
\mathit{divideE}\ m\ n &= \mathbf{do}\ x \leftarrow m \\
&\qquad\qquad y \leftarrow n \\
&\qquad\qquad \mathit{when}\ (y \equiv 0)\ \$\ \mathit{fail}\ \texttt{"division by zero"} \\
&\qquad\qquad \mathit{return}\ \$\ x\ \text{`}\mathit{div}\text{`}\ y
\end{aligned}
$$

Das ist doch kein Zustand!

Wir wollen eine Liste umdrehen und dabei zählen wieviele Nullen sie enthält.

$$
\begin{aligned}
&\textbf{type } \mathit{Count}\ a = \mathit{Int} \to (a, \mathit{Int}) \\
&\mathit{inc} \quad :: \mathit{Count}\ () \\
&\mathit{inc} \quad = \lambda n \to ((), n+1) \\
&\mathit{returnC} \quad :: a \to \mathit{Count}\ a \\
&\mathit{returnC}\ a \quad = \lambda n \to (a, n) \\
&\mathit{bindC} \quad :: \mathit{Count}\ a \to (a \to \mathit{Count}\ b) \to \mathit{Count}\ b \\
&\mathit{bindC}\ m\ k \quad = \lambda n \to \textbf{let } (a, n') = m\ n \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \ k\ a\ n'
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Monad}\ \mathit{Count} \textbf{ where} \\
&\quad \mathit{return} = \mathit{returnC} \\
&\quad (\gg\!=) \ = \mathit{bindC}
\end{aligned}
$$

Haskell meckert...

Zustand

Haskell nötigt uns zu einer **newtype**-Deklaration. Das ist ein **data** mit genau einem Konstruktor, der wiederum genau ein Argument nimmt.

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Count}\ a = \mathit{C}\ (\mathit{Int} \to (a, \mathit{Int})) \\
&\mathit{runC} && :: \mathit{Count}\ a \to \mathit{Int} \to (a, \mathit{Int}) \\
&\mathit{runC}\ (\mathit{C}\ f)\ n && = f\ n \\
&\mathit{inc} && :: \mathit{Count}\ () \\
&\mathit{inc} && = \mathit{C}\ \$\ \lambda n \to ((), n+1) \\
&\mathit{returnC} && :: a \to \mathit{Count}\ a \\
&\mathit{returnC}\ a && = \mathit{C}\ \$\ \lambda n \to (a, n) \\
&\mathit{bindC} && :: \mathit{Count}\ a \to (a \to \mathit{Count}\ b) \to \mathit{Count}\ b \\
&\mathit{bindC}\ m\ k && = \mathit{C}\ \$\ \lambda n \to \textbf{let}\ (a, n') = \mathit{runC}\ m\ n \\
& && \qquad\qquad\quad \textbf{in}\ \ \mathit{runC}\ (k\ a)\ n'
\end{aligned}
$$

instance *Monad Count* **where** ...

Reverse mit Nullenzähler

Die endrekursive Definition von *reverse*, monadisch, mit Zähler:

$$
\begin{aligned}
&\mathit{revApp} && :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Count}\ [\mathit{Int}] \\
&\mathit{revApp}\ [\,] && \mathit{acc} = \mathit{return}\ \mathit{acc} \\
&\mathit{revApp}\ (x : \mathit{xs})\ \mathit{acc} && = \mathbf{do}\ \mathit{when}\ (x \equiv 0)\ \$\ \mathit{inc} \\
& && \qquad \mathit{revApp}\ \mathit{xs}\ (x : \mathit{acc}) \\
&\mathit{revCountZeros} && :: [\mathit{Int}] \to ([\mathit{Int}], \mathit{Int}) \\
&\mathit{revCountZeros}\ \mathit{l} && = \mathit{runC}\ (\mathit{revApp}\ \mathit{l}\ [\,])\ 0
\end{aligned}
$$

Test:

$$
\begin{aligned}
&\mathit{revCountZeros}\ [0,0,1,0,1,2,0,1,2,3] \\
&([3,2,1,0,2,1,0,1,0,0], 4)
\end{aligned}
$$

Parametrisierte Zustands-Monade

Verallgemeinerung von einem *Int*-Zähler auf beliebigen Zustand *s*.

$$\begin{aligned}
&\textbf{newtype}\ State\ s\ a = State\{\, runState :: s \to (a, s)\,\}\\
&returnSt \quad :: a \to State\ s\ a\\
&returnSt\ a \quad = State\ \$\ \lambda s \to (a, s)\\
&bindSt \quad :: State\ s\ a \to (a \to State\ s\ b) \to State\ s\ b\\
&bindSt\ m\ k \quad = State\ \$\ \lambda s \to \textbf{let}\ (a, s') = runState\ m\ s\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ runState\ (k\ a)\ s'\\
&\textbf{instance}\ Monad\ (State\ s)\ \textbf{where}\\
&\quad return = returnSt\\
&\quad (\gg\!=) \ = bindSt
\end{aligned}$$

Eine **data**-Deklaration kann auch gleich Destruktoren mitdefinieren. Im Falle von **newtype** ist es genau ein Destruktor.

$$\begin{aligned}
&runState \quad :: State\ s\ a \to s \to (a, s)\\
&runState\ (State\ f)\ s = f\ s
\end{aligned}$$

IO-Monade

Die Kommunikation zur Außenwelt erfolg in Haskell über die IO-Monade (*input/output*). Zum Beispiel:

instance *Monad IO*

```
putChar     :: Char → IO ()     -- put char. on stdio
putStr      :: String → IO ()   -- put string on stdio
putStrLn    :: String → IO ()   -- put string + newline on stdio
getChar     :: IO Char          -- read char. from stdio
getLine     :: IO String        -- read line from stdio
getContents :: IO String        -- read whole input from stdio
```

Weiteres: Datei-Ein-Ausgabe.

My first I/O-Program in Haskell

Ich habe gestern Nacht ein neues Programm erfunden!

```
main :: IO ()
main = do c ← getChar
          putChar c
          main
```

Ich taufe es *cat*. Auf zum Patentamt.

`http://stopsoftwarepatents.eu/`