Universität Frankfurt/Main
FB 20, Informatik
Wintersemester 1996/97

Seminar:

Parallelisierung und verteilte Ausführung funktionaler Programme
Prof. Dr. Manfred Schmidt-Schauß

Thema:

# DIE VEKTORISIERUNGS-MONADE[1]

von
Heiner Bruns
(e-mail: bruns@stud.uni-frankfurt.de)

Nach einem Forschungspapier von
*Jonathan M. D. Hill, Keith M. Clarke und Richard Bornat*
*Department of Computer Science*
*Queen Mary & Westfield College*
*University of London*

[1] Engl. Originaltitel: *The vectorisation monad*, [6]

**INHALT:**

> *„This improved linear problem can then be transformed into a parallel $\mathcal{O}(\log N)$ algorithm. If $N = 1000$, then the empirical difference is approximately one million compared to ten – not even an implementation in a functional language would have a constant of proportionality that outweighed the improved complexity measure!"*

## 1 Einleitung

In imperativen Programmiersprachen zählen *for*- und *while*-Schleifen zu den wohl am häufigsten genutzten Kontrollstrukturen innerhalb eines Programms. Funktionale Programmiersprachen hingegen kommen sehr gut ohne diese Strukturen aus, da sie sich in den meisten Fällen durch die Standardfunktionen *fold* und *map* bzw. durch Rekursion ausdrücken lassen. Aufgrund dieses Zusammenhanges kann man Schleifenkonstrukte in funktionalen Sprachen als „syntaktischen Zucker" bezeichnen, eine syntaktische Erweiterung also, die die Ausdruckskraft der Sprache unverändert läßt, jedoch die Lesbarkeit der in ihr geschriebenen Programme positiv beeinflussen kann.

Ausgehend von einer solchen „gezuckerten" funktionalen Sprache wird im folgenden der Frage nachgegangen, unter welchen Umständen sich Schleifen erfolgreich, d. h. mit gesteigerter Effizienz gegenüber der sequentiellen Variante, parallelisieren lassen. Hierzu werden die Voraussetzungen hergeleitet und entsprechende Regeln aufgestellt. An einem Beispiel wird man sehen, daß sich mit diesen Regeln einige Probleme auf einem parallelen Rechner effizienter lösen lassen.

Trotz der aufgestellten Regeln ist die Transformation der ursprünglichen Schleife in eine mit Erfolg parallelisierbare Form kein triviales Unterfangen, da sich mit Ihnen auch solche Konstrukte erzeugen lassen die man zwar gefahrlos parallel auswerten kann, was jedoch nicht zwingend zu einer gesteigerten Effizienz führt. Das bedeutet, daß man in unserer „gezuckerten" funktionalen Sprache sowohl solche Schleifen schreiben kann, die sich mit Erfolg parallelisieren lassen, als auch solche bei denen dieser Versuch scheitert.

Ausgehend hiervon wird daraufhin die Technik des monadischen Programmierens dazu benutzt, die Ausdruckskraft der verwendeten funktionalen Sprache so weit einzuschränken, daß sich schließlich nur noch solche Schleifenkonstrukte ohne Typverletzungen schreiben lassen, die dann auch tatsächlich mit Effizienzgewinn parallel ausgewertet werden können.

# 2 Grundlagen

## 2.1 pH: parallel Haskell

parallel Haskell (pH) ist eine parallele Programmiersprachenvariante der funktionalen Sprache Haskell. Im wesentlichen läßt sich pH in drei Schichten einteilen:

- Funktionaler Kern
- I-Struktur-Erweiterungen
- M-Struktur-Erweiterungen

Der funktionale Kern von pH entspricht im wesentlichen Haskell, erweitert um Schleifenkonstrukte:

| | | | | |
|---|---|---|---|---|
| (pattern) | *pat* | ↦ | `next` *var* | Loop variable |
| (expression) | *exp* | ↦ | `while` *exp* do {pat =exp [;]}$^+$ `finally` *exp* | While-loop |
| | | \| | `for` *pat* `<-` *exp* do {pat =exp [;]}$^+$ `finally` *exp* | For-loop |
| | | \| | `next` *var* | Loop variable |

**Abbildung 1: Schleifen in pH**

Semantisch werden pH-*for*-Schleifen in while-Schleifen übersetzt und für diese wiederum wird eine Übersetzung in endrekursive (*tail-recursive*) Funktionen angegeben[2]. Somit handelt es sich bei den Schleifen in pH um eine rein syntaktische Erweiterung von Haskell, die die Ausdruckskraft der Sprache unverändert läßt.

Auf die I-Struktur- und M-Struktur-Erweiterungen werde ich hier nicht eingehen, da pH hier lediglich als syntaktisch gezuckerte Variante von Haskell benutzt wird, um *for*-Schleifen programmieren zu können.

## 2.2 Parallele Algorithmen für fold und scan

Wie bereits in der Einleitung dargestellt, soll es zunächst darum gehen, in pH geschriebene *for*-Schleifen zu parallelisieren. Dafür ist es hilfreich, zunächst einmal ein Beispiel für ein paralleles Rechnermodell zu betrachten, und sich mit der Frage auseinanderzusetzen, wie sich die Standard Haskell-Funktionen *fold* und *scan* auf einem solchen Rechner parallelisieren lassen, bzw. welchen Effizienzgewinn man durch die Parallelisierung erhält[3].

### 2.2.1 Ein paralleles Rechnermodell

Das im folgenden als Beispiel benutzte parallele Rechnermodell ist das häufig verwendete Modell der PRAM (***P****arallel* ***R****andom* ***A****ccess* ***M****achine*) wie es u. a. auch in [3] beschrieben und eingesetzt wird.

Eine PRAM besteht aus $p$ Prozessoren $P_0, P_1, \ldots, P_{p-1}$ die mit einem gemeinsam genutzten, beliebig großem Speicher (*shared memory*) verbunden sind. Jeder Prozessor kann auf ein beliebiges Datum in diesem Speicher in konstanter Zeit zugreifen (es können auch alle

[2] Für eine formale Darstellung dieser Übersetzungen s. [7], S. 2 ff.

[3] Wir beschäftigen uns hier ganz gezielt lediglich mit der Parallelisierung der Funktionen *scan* und *fold*, da dies - wie wir später sehen werden - für unsere Zwecke vollkommen ausreichend ist.

Prozessoren gleichzeitig einen Speicherzugriff durchführen). Die Laufzeit eines parallelen Algorithmus hängt somit nicht mehr allein von der Eingabegröße $N$ des Problems ab, sondern zusätzlich von der Anzahl $p$ der verwendeten Prozessoren. Als kleines Beispiel hierzu kann man sich die Haskell-Funktion *map* für Listen betrachten:

```
map            :: (α → β) → [α] → [β]
map f []       = []
map f (x:xs) = f x : map f xs
```

Vorausgesetzt, daß es sich bei f um eine in konstanter Zeit berechenbare Funktion handelt ist eine sequentielle Implementierung der Funktion *map* bestenfalls in $\mathcal{O}(N)$ möglich, wobei $N$ die Länge der Liste ist. Eine parallele Implementierung ist hingegen auf einer PRAM mit $p = N$ Prozessoren in $\mathcal{O}(1)$ möglich.

Ein PRAM-Algorithmus bei dessen Ausführung mehrere Prozessoren gleichzeitig den Inhalt ein und derselben Speicherzelle lesen können nennt man einen *concurrent-read* Algorithmus. Ein *exclusive-read* Algorithmus hingegen ist ein PRAM-Algorithmus, bei dem solche gleichzeitigen Lesezugriffe auf dieselbe Speicherzelle nicht gestattet sind und zu einem Fehler führen. Völlig analog unterscheiden sich *concurrent-write* von *exclusive-write* Algorithmen. Insgesamt gibt es demnach vier verschiedene Typen von PRAM-Algorithmen:

- EREW: *exclusive read* und *exclusive write*
- CREW: *concurrent read* und *exclusive write*
- ERCW: *exclusive read* und *concurrent write*
- CRCW: *concurrent read* und *concurrent write*

Die populärsten dieser Algorithmen sind die beiden Extreme EREW und CRCW. Zu beachten ist dabei, daß ein für eine EREW-PRAM entwickelter Algorithmus problemlos auch auf einer CRCW-PRAM ausgeführt werden kann, nicht jedoch umgekehrt. EREW-Algorithmen zählen somit zu den vielseitigsten PRAM-Algorithmen, da sie auf jeder PRAM erfolgreich eingesetzt werden können. Ein solcher Algorithmus soll im folgenden betrachtet werden.

### 2.2.2 Pointer jumping

In diesem Abschnitt möchte ich einen Algorithmus, bekannt als *Pointer jumping* vorstellen, mit dem eine effiziente parallele Implementierung der Haskell-Funktionen *fold* und *scan* möglich ist. Betrachten wir hierzu zunächst eine informelle Darstellung dieser beiden Funktionen[4]:

$$\texttt{foldl}\ (\oplus)\ \texttt{a}\ \left[x_1, x_2, \ldots, x_n\right] = \left(\ldots\left(\left(a \oplus x_1\right) \oplus x_2\right)\ldots\right) \oplus x_n$$

$$\texttt{scan}\ (\oplus)\ \texttt{a}\ \left[x_1, x_2, \ldots, x_n\right] = \left[a, a \oplus x_1, \left(a \oplus x_1\right) \oplus x_2, \ldots\right]$$

Hierbei fällt auf, daß für endliche Listen der folgende Zusammenhang gilt[5]:

[4] Ich beschränke mich hier auf die *fold*-Variante *foldl* (*fold-left*). Der Zusammenhang zwischen *foldl* und *foldr* (*fold-right*) wird an anderer Stelle dieser Arbeit noch betrachtet.

[5] *last* ist die Funktion, die das letzte Element einer als Argument übergebenen Liste zurückliefert. Die Funktion *(.)* ist die Funktionskomposition: $(.)\ f\ g = \lambda x \to f\left(g\ x\right)$

```
foldl (⊕) a = last . scan (⊕) a
```

Somit genügt es im folgenden, eine parallele Implementierung für *scan* zu betrachten. Eine sequentielle Implementierung von *scan* auf einer Liste mit $N$ Elementen ist bestenfalls in $\mathcal{O}(N)$ möglich. Vorausgesetzt wird hier, daß die Funktion $(\oplus)$ $\mathcal{O}(1)$ Zeit benötigt und darüber hinaus assoziativ ist. Der nachfolgend vorgestellte parallele Algorithmus löst dieses Problem auf einer EREW-PRAM mit $p = N$ Prozessoren in $\mathcal{O}(\log N)$ Zeit.

Zunächst ein Wort zur verwendeten Notation: $y[i]$ bezeichne den Wert des i-ten Listenelements der Liste L. L selbst sei eine einfach verkettete Liste und $next[i]$ sei ein Zeiger auf das dem i-ten Listenelement nachfolgende Element ($next[N] = NIL$). Die Schreibweise $a \leftarrow b$ bedeutet, daß die Speicherzelle $a$ den Wert zugewiesen bekommt, der in Speicherzelle $b$ steht. Der folgende Pseudocode beschreibt nun den eigentlichen Algorithmus[6]:

```
while there exists an object i such that next[i] ≠ NIL
  do for each processor i, in parallel
       do if next[i] ≠ NIL
            then y[next[i]] ← y[i] ⊕ y[next[i]]
                 next[i] ← next[next[i]]
```

Die nachfolgende Abbildung veranschaulicht die Arbeitsweise dieses Algorithmus[7]. Die verwendete Funktion $(\oplus)$ ist in diesem Beispiel die Multiplikation. Die einzelnen dargestellten Schritte (a) bis (d) zeigen jeweils die Liste vor der Ausführung der **while**-Schleife:

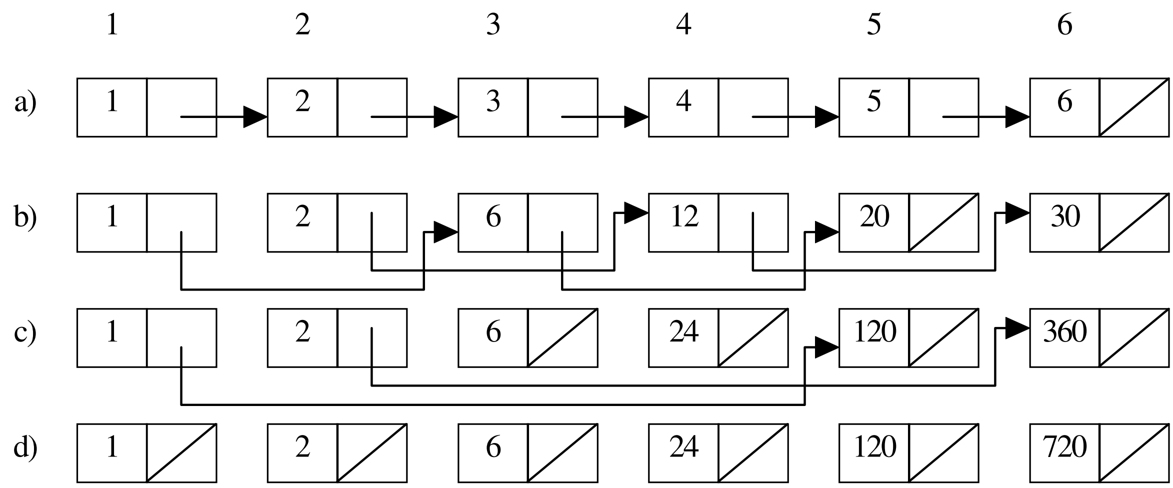


**Abbildung 2: Veranschaulichung der Arbeitsweise des Pointer jumping Algorithmus**

Ich denke, man kann sich bei diesem Algorithmus relativ leicht von der logarithmischen Laufzeit überzeugen. Für einen formalen Beweis der *worst-case* Laufzeit $\mathcal{O}(\log N)$ dieses Algorithmus schlage man in [3] nach.

[6] Es wird davon ausgegangen, daß jeweils ein Prozessor i für die Bearbeitung des i-ten Listenelements zuständig ist. D. h. es gibt mindestens soviele Prozessoren wie die Anzahl der Elemente in der zu bearbeitenden Liste.

[7] Aus der Abbildung wird auch deutlich, warum dieser Algorithmus *Pointer jumping* genannt wird.

Wir haben nun gesehen, daß man in einer parallelen Rechnerumgebung ein *fold* auf einer Liste mit $N$ Elementen und einem assoziativen Operator $\oplus$ in $\mathcal{O}(\log N)$, gegenüber $\mathcal{O}(N)$ in einer sequentiellen Rechnerumgebung durchführen kann. Weiterhin wissen wir, daß ein paralleles *map* in $\mathcal{O}(1)$ realisierbar ist. Im folgenden geht es nun um die Parallelisierung von in der Syntax der Sprache pH geschriebenen *for*-Schleifen.

# 3 Parallelisierung einfacher Schleifen

## 3.1 Wann ist eine Parallelisierung möglich?

Wie bereits erwähnt, lassen sich *for*-Schleifen in Programmen problemlos in Form von endrekursiven Funktionen ausdrücken. In [7] befindet sich ein entsprechendes Übersetzungsschema für in pH geschriebene *for*-Schleifen in entsprechende endrekursive Funktionen. Am konkreten Beispiel einer in pH geschriebenen Fakultätsfunktion sieht diese Übersetzung wie folgt aus:

```
fact n = let acc = 1
         in for i <- [1..n] do
              next acc = acc * i
            finally acc
```

```
fact n = f 1 [1..n]
         where
           f acc []     = acc
           f acc (i:is) = f (acc * i) is
```

Dieser Zusammenhang ist zwar interessant, jedoch leider zum Zwecke der Parallelisierung einer Schleife unbrauchbar, denn alles was wir erhalten haben ist eine sequentiell abzuarbeitende endrekursive Funktion. Dennoch ist interessant zu beobachten, daß diese Funktion im Prinzip nichts weiter ist, als eine Instanz von *fold-left* über den Bereich der *for*-Schleife:

```
fact 3 = f 1 [1..3]
       = f (1 * 1) [2..3]
       = f ((1 * 1) * 2) [3]
       = f (((1 * 1) * 2) * 3) []
       = (((1 * 1) * 2) * 3)
       = ...
```

Die *next*-Funktion ist die Funktion, die bei einem einzelnen Iterationsschritt einer *for*-Schleife in deren Rumpf berechnet wird. Die oben dargestellte Fakultätsfunktion kann somit auch als ein *fold-left* der *next*-Funktion über den Bereich der *for*-Schleife geschrieben werden:

```
fact n = foldl (\acc i -> acc * i) 1 [1..n]
```

Das im folgenden dargestellte Übersetzungsschema TPH beschreibt die Übersetzung von pH *for*-Schleifen in ein *fold-left* im allgemeinen Fall:

$$\mathsf{TPH}\left[\!\!\left[\begin{array}{l}\texttt{for i <-}\mathit{range}\ \texttt{do}\\ \quad \mathit{pat}_1 \ = \ \mathit{exp}_1\\ \quad \vdots \qquad \vdots\\ \quad \mathit{pat}_n \ = \ \mathit{exp}_n\\ \texttt{finally}\ \mathit{exp}_{fin}\end{array}\right]\!\!\right] = \texttt{case}\left(\texttt{foldl}\left(\begin{array}{l}\lambda\ (s_1,\ldots,s_m)\ i \rightarrow\\ \quad \texttt{let}\ \mathsf{TPAT}[\![\mathit{pat}_1]\!] = \mathsf{TPH}[\![\mathit{exp}_1]\!]\\ \qquad\qquad \vdots \qquad\qquad \vdots\\ \qquad\quad \mathsf{TPAT}[\![\mathit{pat}_n]\!] = \mathsf{TPH}[\![\mathit{exp}_n]\!]\\ \quad \texttt{in}\ (v'_1,\ldots,v'_m)\end{array}\right)\ (v_1,\ldots,v_m)\ \mathit{range}\right)\ \texttt{of}$$

$$\sim(v_1,\ldots,v_m)\ \texttt{->}\ \mathit{exp}_{fin}$$

$$\text{where } \{(v_1,v'_1),\ldots,(v_m,v'_m)\} = \{(\texttt{v},\texttt{v'})\mid \texttt{next v} \in \{\mathit{pat}_1,\ldots \mathit{pat}_n\}\}$$

$\mathsf{TPH}[\![\texttt{next var}]\!] = \texttt{var'}$ where $\mathsf{TPH}[\![\mathit{exp}]\!]$ is the identity translation for other expressions.

$\mathsf{TPAT}[\![\texttt{next var}]\!] = \texttt{var'}$where $\mathsf{TPAT}[\![\mathit{pat}]\!]$ is the identity translation for other patterns.

**Abbildung 3: Übersetzung einer pH for-Schleife in ein fold-left**

Nach dem ersten Dualitätstheorem [2] gilt der Zusammenhang

$$foldl\,(\otimes)\,x\,xs \equiv foldr\,(\otimes)\,x\,xs \equiv foldPar\,(\otimes)\,x\,xs$$

**Gleichung 1: Das erste Dualitätstheorem**

unter den folgenden beiden Voraussetzungen:

- $(\otimes)$ und $x$ bilden ein Monoid[8]
- $xs$ ist eine endliche Liste

Damit lassen sich pH *for*-Schleifen genau dann mit Erfolg parallelisieren, wenn die *next*-Funktion assoziativ ist und gemeinsam mit dem initialen Wert der Schleife ein Monoid bildet. Dann kann das *fold-left* durch ein paralleles *fold* (*foldPar*) ersetzt werden und wir haben eine Laufzeit von $\mathcal{O}(\log N)$ falls die Laufzeit vor der Transformation $\mathcal{O}(N)$ war. Wie ein solches paralleles *fold* funktioniert wurde bereits gezeigt.

Die Zusatzbedingung, daß die Liste endlich sein muß wird im folgenden ignoriert, da in allen Fällen, in denen ein *fold-left* ein wohldefiniertes[9] Ergebnis liefert, ein *fold-right* dasselbe Ergebnis berechnet.

Es soll nun untersucht werden, ob die Assoziativität der *next*-Funktion für eine erfolgreiche Parallelisierung wirklich notwendig ist, oder ob sich auch solche Schleifen parallelisieren lassen, bei denen sie nicht assoziativ ist.

## 3.2 Ist Assoziativität unbedingt erforderlich?

Die Fähigkeit *for*-Schleifen parallelisieren zu können, bei denen die *next*-Funktion assoziativ ist, ist leider in der Praxis nicht sonderlich brauchbar, da aufgrund der Assoziativität der *next*-Funktion beide Argumente und das Ergebnis vom selben Typ sein müssen. Hat man eine *next*-Funktion der Form $s' = s \otimes i$, so verkörpert $s$ den aktuellen Zustand der Schleife, $i$ den Schleifenzähler und $s'$ den Nachfolgezustand. In herkömmlichen numerischen Algorithmen,

[8] Das ist der Fall, wenn der Operator $\otimes$ assoziativ ist und $x$ das neutrale Element bzgl. der Operation $\otimes$ ist. (Monoid = Halbgruppe)

[9] Wohldefiniert bedeutet hier, daß die Funktion ein Ergebnis liefert welches nicht $\perp$ ist.

die einen großen Teil der Funktionen die wir parallelisieren möchten ausmachen, ist $i$ in der Regel ein ganzzahliger Typ und $s$ bzw. $s'$ sind Fließkommazahlen.

In den Fällen, in denen die zu parallelisierende Schleife nur einen Zustand hat, also wenn beispielsweise nur eine mit *next* markierte Variable im Schleifenrumpf vorkommt, besteht eine Lösung dieses Problems darin, die *next*-Funktion in zwei Teile zu zerlegen: einen für die Berechnung des Schleifenzählers und einen weiteren für die Berechnung des Schleifenzustands:

$$\lambda \; s \; i \;\; \rightarrow \qquad s \;\oplus \qquad f \quad i$$

wobei hier wiederum der initiale Wert der *for*-Schleife zusammen mit dem Operator $\oplus$ ein Monoid bilden muß. Das folgende *fold-map fusion law* [1] definiert nun, wie man mit Hilfe dieser Zerlegung einer nicht-assoziativen *next*-Funktion, eine entsprechende *for*-Schleife dennoch erfolgreich parallelisieren kann:

$$\begin{aligned} foldl\,\overbrace{\left(\lambda s\ i \rightarrow s \oplus f\ i\right)}^{next-Funktion}\,x\ xs &\equiv \left(\underbrace{foldl\,(\oplus)\,x}_{g} \circ \underbrace{map\ f}_{h}\right) xs \\ &\equiv \left(\lambda y \rightarrow \underbrace{\left(foldl\,(\oplus)\,x\right)}_{g}\left(\underbrace{\left(map\ f\right)}_{h}\,y\right)\right) xs \\ &\equiv foldl\,(\oplus)\,x\left(map\ f\ xs\right) \end{aligned}$$

**Gleichung 2: Das fold-map fusion law**

Nach dieser Umformung sind wieder alle Voraussetzungen für eine Verwendung von *foldPar* anstelle von *foldl* gegeben. Der zurückliegende Abschnitt hat gezeigt, daß die ursprüngliche *next*-Funktion der *for*-Schleife nicht unbedingt assoziativ sein muß um die Schleife parallel berechnen zu können. Dennoch sind in den durch das *fold-map fusion law* erlaubten Umformungen auch einige Gefahren verborgen, wie der nächste Abschnitt zeigen wird.

### 3.3 Das „Leaky fold-left law"

Ein bekanntes Problem nicht-strikter funktionaler Sprachen wie Haskell ist der hohe Speicherplatzbedarf für die Auswertung bestimmter Ausdrücke (z. B. *foldl*), hervorgerufen durch die outermost reduction order oder lazy-evaluation. Betrachten wir hierzu das folgende Beispiel aus [2]:

```
sum = foldl (+) 0
sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) ((0+1)+2) [3..1000]
= ...
= foldl (+) (...((0+1)+2)+...+1000) []
= (...((0+1)+2)+...+1000)
= 500500
```

Wie man sieht wird hier aufgrund der outermost reduction order zunächst eine Hülle aufgebaut, die Speicherplatz linear zur Eingabegröße benötigt. Erst nachdem diese Hülle vollständig aufgebaut wurde, wird mit der eigentlichen Berechnung der Funktion begonnen.

Dieses Problem wird ein intelligenter Compiler bzw. Interpreter mit Sicherheit umgehen, indem er für einen solchen Ausdruck eine sinnvolle Mischung aus innermost und outermost reduction order verwendet:

```
sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) 1 [2..1000]
= foldl (+) (1+2) [3..1000]
= foldl (+) 3 [3..1000]
= ...
= foldl (+) 500500 []
= 500500
```

In dem vorgestellten Beispiel läßt sich die Vergeudung von Speicherplatz durch geschickte Auswertung vermeiden. Jedoch ist es auch möglich, die Bildung dieser Hülle linearer Größe bei *fold-left* zu erzwingen – in strikten funktionalen Sprachen ebenso wie in nicht-strikten funktionalen Sprachen. Das ist das sog. *leaky fold-left law*:

$$\mathit{foldl}\,(\oplus)\ x\ \ xs \equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow (\lambda y \rightarrow y \oplus i) \circ s\right)\ id\ \ xs\right) x$$

**Gleichung 3: Das leaky fold-left law**

Bevor ich auf diesen Zusammenhang etwas genauer eingehe, möchte ich zunächst einmal die obige Gleichung ein wenig umformen[10]:

$$\begin{aligned}
\mathit{foldl}\,(\oplus)\ x\ \ xs &\equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow (\lambda y \rightarrow y \oplus i) \circ s\right)\ id\ \ xs\right) x \\
&\equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow \left(\lambda y \rightarrow i \mathbin{\tilde{\oplus}} y\right) \circ s\right)\ id\ \ xs\right) x \\
&\equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow s \mathbin{\tilde{\circ}} \left(\lambda y \rightarrow i \mathbin{\tilde{\oplus}} y\right)\right)\ id\ \ xs\right) x \\
&\equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow s \mathbin{\tilde{\circ}} \left(\left(\tilde{\oplus}\right) i\right)\right)\ id\ \ xs\right) x \\
&\equiv \left(\mathit{foldl}\left(\lambda s\ i \rightarrow s \mathbin{\tilde{\circ}} \left(\left(\tilde{\oplus}\right) i\right)\right)\ id\ \ xs\right) x \\
&\equiv \left(\mathit{foldl}\,(\tilde{\circ})\ id\ \left(\mathit{map}\ \left(\tilde{\oplus}\right)\ xs\right)\right)\ x
\end{aligned}$$

**Gleichung 4: Das leaky fold-left law als eine Instanz des fold-map fusion laws**

Was bei dieser Umformung herausgekommen ist, ist eine Instanz des *fold-map fusion laws*. Da der Operator $\tilde{\circ}$ assoziativ ist und gemeinsam mit der Identitätsfunktion id ein Monoid bildet, kann das *fold-left* in der obigen Gleichung durch ein paralleles *fold* ersetzt werden. Diese Umformung könnten wir theoretisch auf jede *next*-Funktion einer *for*-Schleife, vollkomen unabhängig von ihrer Struktur anwenden. Die Schleife ließe sich dann in jedem Fall parallelisieren. Wenn man sich jedoch einmal genauer ansieht, was bei einer Auswertung der obigen Instanz des *fold-map fusion laws* geschieht, wird recht schnell deutlich, daß das Ersetzen von *foldl* durch *foldPar* keine Effizienzsteigerung bewirken wird:

[10] $\tilde{\oplus}$ bezeichne im folgenden den Operator $\oplus$ mit vertauschten Argumenten, also $(\tilde{\oplus}) : \lambda x\ \ y \rightarrow y \oplus x$. Gleiches gilt für den Zusammenhang zwischen $\circ$ und $\tilde{\circ}$.

1. $\tilde{\oplus}$ wird auf jedes einzelne Element der Liste xs angewendet (berechnet werden kann hier nichts; aufgrund fehlender Argumente).

2. Auf die resultierende Liste wird *fold* mit dem *fold*-Operator $\tilde{\circ}$ angewendet. Dadurch wird eine Hülle, linear zur Länge der Liste aufgebaut (berechnet werden kann auch hier nichts; wiederum aufgrund fehlender Funktionsargumente). Hatte die ursprüngliche Liste die Form $[1,2,3]$, so sieht die resultierende Hülle folgendermaßen aus: $(\oplus 3) \circ (\oplus 2) \circ (\oplus 1)$.

3. Erst jetzt kann die eigentliche Berechnung beginnen, denn erst zu diesem Zeitpunkt kann die bisher aufgebaute Hülle auf den initialen Wert x angewendet werden.

Die eigentliche Berechnung der Funktion findet also in Schritt 3 statt. Parallelisieren würden wir durch Verwendung von *foldPar* anstelle von *foldl* allerdings Schritt 2. Die gewünschte Effizienzsteigerung und die damit verbundene logarithmische Laufzeit der Schleife kann auf diesem Wege also nicht erreicht werden, obwohl es auf den ersten Blick so schien.

Was bisher gezeigt wurde ist, daß eine *for*-Schleife nichts weiter als syntaktischer Zucker für ein *fold-left* ist. Ist die von diesem *fold-left* verwendete Funktion assoziativ, so läßt sich das *fold-left* durch ein paralleles *fold* ersetzen und somit erfolgreich parallelisieren. Ist sie nicht assoziativ, so besteht dennoch eine Chance für eine erfolgreiche Parallelisierung, vorausgesetzt wir finden eine dem *fold-map fusion law* entsprechende Zerlegung die letztendlich nicht auf eine Version des *leaky fold-left law* herausläuft. Genau in diesem Punkt besteht die Schwierigkeit des bisher vorgestellten Ansatzes: Wir können nicht vorhersehen, ob sich eine programmierte Schleife mit Effizienzgewinn parallelisieren lassen wird oder nicht. Daher wird später ein Verfahren basierend auf der Technik des monadischen Programmierens dazu benutzt, die Ausdruckskraft der verwendeten funktionalen Sprache so weit einzuschränken, daß sich schließlich nur noch solche Schleifenkonstrukte ohne Typverletzungen schreiben lassen, die dann auch tatsächlich erfolgreich parallelisiert werden können. Doch zunächst soll ein kleines Beispiel für die Parallelisierung einer pH *for*-Schleife mit den bisher kennengelernten Mitteln vorgestellt werden.

# 4 Beispiel für eine Parallelisierung

## 4.1 Der Primzahltest

Die folgende in pH geschriebene Funktion testet, ob das ihr übergebene Argument eine Primzahl ist:

```
isPrime :: Int -> Bool
isPrime n
  = let s = True
    in for i <- [2 .. truncate (sqrt (fromInteger n))]
        do
         next s = if n `rem` i == 0
                  then False
                  else s
       finally s
```

Der erste Schritt bei der Parallelisierung dieser Schleife besteht darin festzustellen, ob der Rumpf der *for*-Schleife eine assoziative Funktion darstellt. Wenn ja, dann gäbe es keine weiteren Probleme. Leider ist das jedoch nicht der Fall, da der Schleifenrumpf eine Bedingung enthält. Mit der folgenden Äquivalenz zwischen der hier verwendeten Bedingung und einer logischen Konjunktion ist das Problem jedoch in den Griff zu bekommen:

$$if\ x\ then\ \mathrm{False}\ else\ y \equiv \neg x \wedge y$$

Die Schleife läßt sich dann in die folgende Instanz von *foldl* übersetzen:

```
isPrime n
  = foldl (\s i -> not (n `rem` i == 0) && s) True
          [2 .. truncate (sqrt (fromInteger n))]
```

Die *next*-Funktion hat nun eine Form die es gestattet, das *fold-map fusion law* anzuwenden:

$$\begin{aligned} \lambda s\ i \to \neg\left(n\ rem\ i = 0\right) \wedge s \equiv \lambda s\ i \to s \oplus f\ i \\ \mathit{mit}\ \ f\ i = \neg\left(n\ rem\ i = 0\right) \\ \mathit{und}\ \ s \oplus t = t \wedge s \end{aligned}$$

Wendet man das *fold-map fusion law* nun an, so erhält man schließlich:

```
isPrime n
  = foldl (\ s t -> t && s) True
          (map (\i -> n `rem` i /= 0)
               [2 .. truncate (sqrt (fromInteger n))])
```

Wie man sieht sind nun sämtliche Voraussetzungen für eine Anwendung des ersten Dualitätstheorems gegeben, und der Ersetzung des *foldl* durch ein *foldPar* steht nichts mehr im Wege. Während die ursprüngliche pH-Funktion eine Laufzeit von $\mathcal{O}\left(\sqrt{n}\right)$ hatte, hat die parallelisierte Version eine Laufzeit von $\mathcal{O}\left(\log \sqrt{n}\right)$.

### *4.2 Weiterführende Überlegungen*

Was bisher noch nicht angesprochen wurde ist der Fall, daß eine *for*-Schleife in ihrem Rumpf mehrere Zustände, also mehrere durch *next* markierte Variablen aufweist. Ich möchte im folgenden nur ganz kurz skizzieren, was man in einem solchen Fall tun kann. Für eine ausführliche Darstellung sowie ein Beispiel hierzu sei auf [5] bzw. [6] verwiesen.

Das Ziel soll sein, am Ende eine einzige mit *next* markierte Variable im Rumpf der Schleife zu haben. Um das zu erreichen, werden alle anderen mit *next* markierten Variablen aus dem Rumpf der Schleife herausgenommen, d. h. bereits vor der eigentlichen Schleifenausführung berechnet. Dabei ist es natürlich wesentlich, nicht nur das Endergebnis dieser Variablen, sondern auch die Zwischenergebnisse aller Iterationsschritte zur Verfügung zu haben. Schließlich wird die eigentliche Schleife diese Zwischenergebnisse zur Berechnung des Gesamtergebnis evtl. benötigen. Würde man diese „Vorausberechnungen" also mit einem *fold* durchführen, so hätte man lediglich das Endergebnis, und keine Zwischenergebnisse. Eine Berechnung mit scan hingegen liefert die komplette Liste der Variablenwerte aller Iterationsschritte. In [2] wird eine Programmidentität angegeben, mit deren Hilfe sich das *fold-map fusion law* in ähnlicher Form auch auf scan anwenden läßt. Das erste Dualitätstheorem [2] ist ebenfalls auf scan anwendbar. Wie wir bereits in gesehen haben, läßt sich für scan eine parallele Version ebenso angeben wie für *fold*. Daher können die Vorausberechnungen der Variablenwerte mit scanPar im Vorfeld der eigentlichen Schleife berechnet und in einer Liste gespeichert werden.

Bleibt im wesentlichen nur noch die Frage zu klären, wie innerhalb der Schleife auf diese Werte zugegriffen werden kann. Dies erreichen wir dadurch, daß wir nun als „Schleifenzähler" keine einfache Liste von Integerwerten mehr verwenden, sondern eine Liste bestehend aus n-tupeln, zusammengesetzt aus dem eigentlichen Schleifenzähler und dem jeweiligen zuvor berechneten Variablenwert zu diesem Iterationsschritt. Eine solche Liste kann durch einfaches, parallel in konstanter Zeit durchzuführendes „zippen" der zuvor berechneten Listen erfolgen.

Das ist eigentlich schon die ganze Idee. Natürlich gibt es hier und da noch einige Schwierigkeiten und Dinge die es zu beachten gilt, aber wie bereits anfangs erwähnt möchte ich in diesen Punkten auf die angegebenen Referenzen verweisen.

# 5 Die Vektorisierungs-Monade

Von nun an geht es darum, eine Technik zu entwickeln die es ermöglicht, nur noch solche Schleifen zu programmieren, die sich problemlos parallelisieren lassen. Dazu beschäftigen wir uns im folgenden mit Monaden.

## 5.1 Monaden in der funktionalen Programmierung[11]

In der funktionalen Programmierung werden Monaden in der Regel benutzt, um imperative, mit Seiteneffekten behaftete Elemente, in einer reinen funktionalen Sprache verwenden zu können, ohne dabei auf die referentielle Transparenz[12] verzichten zu müssen.

Für unsere Zwecke besteht eine Monade aus einem Typkonstruktor M und zwei Funktionen *unit* und *bind* welche die folgenden Typen haben[13]:

$$unit :: \alpha \rightarrow M\ \alpha$$
$$bind :: M\ \alpha \rightarrow (\alpha \rightarrow M\ \beta) \rightarrow M\ \beta$$

Damit das Tripel $(M, unit, bind)$ als Monade bezeichnet werden darf müssen die folgenden drei Bedingungen erfüllt sein:

$$unit\ v\ 'bind'\ (\lambda w \rightarrow k\ w) \equiv k\ v$$
$$m\ 'bind'\ (\lambda v \rightarrow unit\ v) \equiv m$$
$$(m\ 'bind'\ (\lambda v \rightarrow k\ v))\ 'bind'\ (\lambda w \rightarrow h\ w) \equiv m\ 'bind'\ (\lambda v \rightarrow (k\ v\ 'bind'\ (\lambda w \rightarrow h\ w)))$$

Vereinfacht ausgedrückt läßt sich mit den drei Elementen des Tripels $(M, unit, bind)$ folgendes assoziieren:

- Der Typ $M\,\alpha$ repräsentiert irgendeine Berechnung, die einen Wert vom Typ $\alpha$ hervorbringt.
- Die Funktion *unit* wird benötigt, um Werte in Berechnungen einzubeziehen: wenn $v :: \alpha$ ein Wert ist, dann repräsentiert $unit\ v :: M\,\alpha$ eine Berechnung die „nichts“ bewirkt, außer genau den Wert $v$ zu liefern.

---

[11] Vgl. [4] für eine ausführlichere Darstellung von Monaden aus der Sichtweise der Kategorientheorie.

[12] Referentielle Transparenz bedeutet u. a., daß eine Funktion stets denselben Wert zurückliefert, wenn man sie mit denselben Argumenten aufruft. Variablen werden direkt mit Werten assoziiert, und nicht mit Positionen im Hauptspeicher eines Rechners. Daher lassen sich wenn referentielle Transparenz gegeben ist, Ausdrücke mit gleichem Wert beliebig gegeneinander austauschen, ohne Konsequenz für den Wert des Gesamtausdrucks (Substitutionsprinzip).

[13] Anstelle der hier angeführten Definition findet man gelegentlich auch eine äquivalente Definition von Monaden, die anstelle der Funktion bind die beiden Funktionen $map :: (\alpha \rightarrow \beta) \rightarrow (M\,\alpha \rightarrow M\,\beta)$ und $join :: M(M\,\alpha) \rightarrow M\,\alpha$ verwenden. Für unsere Zwecke ist die angegebene Definition jedoch praktischer.

- Die Funktion *bind* dient zur Kombination von zwei Berechnungen, wobei die zweite Berechnung von einem Wert abhängen kann, den die erste liefert: wenn $m :: M\,\alpha$ eine Berechnung ist, und $k :: \alpha \to M\,\beta$ eine Funktion von Werten $\alpha$ nach Berechnungen $M\,\beta$, dann repräsentiert $m\,'bind'\,k :: M\,\beta$ die Berechnung, welche die Berechnung m ausführt, anschließend k auf das Ergebnis anwendet, und schließlich die resultierende Berechnung durchführt.

Im weiteren Verlauf wird nun als Beispiel eine konkrete Monade zur simplen Ausgabe von Text vorgestellt die, wie sich später herausstellen wird, eine Vektorisierungs-Monade[14] ist.

## 5.2 Eine Vektorisierungs-Monade für die Textausgabe

### 5.2.1 Schritt 1: Die ursprüngliche Vektorisierungs-Monade

Es soll nun eine Monade vorgestellt werden die zur Ausgabe von Text (einer Liste von Zeichen), verwendet werden kann. Betrachten wir dazu die folgenden Definitionen in Haskell:

```
type IO a = String -> (a,String)

unit :: a -> IO a
unit x = \s -> (x,s)

bind :: IO a -> (a -> IO b) -> IO b
l `bind` r = \s -> let (res,s') = l s
                   in r res s'

pprint :: String -> IO ()
pprint str = \s -> ( (), s ++ str)
```

*pprint* ist hier eine Funktion, die ihr Argument *str* vom Typ *String* als „Seiteneffekt" auf dem Ausgabe-Stream *s* ausgibt, und *()* (das Null-Tupel) als Ergebnis liefert. $(IO, unit, bind)$ bildet eine Monade, da die obigen Definitionen den drei Gesetzen für Monaden genügen.

Eine monadische *for*-Schleife wird folgendermaßen definiert:

$$\text{for } ctr \leftarrow range \text{ do } body$$

wobei *ctr* die Schleifenvariable ist, *range* der Bereich der Schleife und *body* der Schleifenrumpf. Der Schleifenrumpf ist eine Berechnung vom Typ *IO ()*. Das Ergebnis einer monadischen *for*-Schleife ist stets *()*, aber ihr Zustand ändert sich durch Seiteneffekte während der Ausführung der Schleife. Der momentane Zustand einer monadischen *for*-Schleife ist also im Prinzip im Innern der Monade verborgen, und wird nicht explizit durch mit *next* markierte Variablen angegeben, wie es bei *for*-Schleifen in pH der Fall ist. Die folgenden Zeilen zeigen, wie eine monadische *for*-Schleife im Falle unserer Monade aussehen kann:

[14] Was man genau unter einer Vektorisierungs-Monade zu verstehen hat, wird sich später zeigen. Bisher ist es zunächst nur ein Objekt, welches den beschriebenen Anforderungen an eine Monade genügt.

```
helloWorlds :: Int -> IO ()
helloWorlds n
  = for i <- [1..n] do
      pprint "Hello World " `bind` (\ () ->
      pprint (show i)       `bind` (\ () ->
      pprint "\n"           `bind` (\ () ->
      unit ()
      )))
```

Ähnlich wie *for*-Schleifen in pH lassen sich auch die monadischen *for*-Schleifen mittels *fold-left* ausdrücken. Dies geschieht wie folgt:

1.) Der Schleifenrumpf wird von einer Funktion, welche die Schleifenvariable als Parameter hat modelliert.
2.) Ein *fold-left* der Funktion $\lambda\, s\ i \rightarrow s\ 'bind'\ \left(\lambda() \rightarrow bodyFn\ i\right)$ wird durchgeführt (bodyFn ist dabei die Funktion, die den Rumpf der Schleife modelliert.)
3.) Während des ersten Iterationsschrittes kapselt das Argument *s* des Lambda-Ausdruckes auf den *fold-left* angewendet wird, den initialen Zustand der Schleife.
4.) Während des k-ten Iterationsschrittes kapselt *s* den Zustand aller vorhergehenden k-1 Iterationsschritte der Schleife.
5.) Der initiale Zustand der Schleife ist eine Berechnung die „nichts" tut, also *unit ()*.

Übersetzt man die angegebene *for*-Schleife nun gemäß den obigen Regeln in ein *fold-left*, so erhält man folgendes:

```
helloWorlds :: Int -> IO ()
helloWorlds n
  = foldl (\s i -> s `bind` (\ _ -> bodyFn i))
          (unit ())
          [1..n]
  where
    bodyFn :: Int -> IO ()
    bodyFn i
      = pprint "Hello World " `bind` (\ () ->
        pprint (show i)       `bind` (\ () ->
        pprint "\n"           `bind` (\ () ->
        unit ()
        )))
```

Das Problem welches es jetzt noch zu lösen gilt ist, daß die vom *fold-left* verwendete Funktion nicht assoziativ ist. Dies wird im folgenden dadurch gelöst werden, indem aus der Monade $(IO, unit, bind)$ das Monoid $(IO', \oplus, \oplus_{id})$ gemacht wird. $\oplus$ ist dabei die assoziative Funktion die *bind* ersetzt, und $\oplus_{id}$ das neutrale Element bzgl. dieser Funktion, welches *unit* ersetzt.

Wenn es gelingt, die ursprüngliche Monade in ein solches Monoid umzuformen, dann bezeichnet man sie als Vektorisierungs-Monade.

### 5.2.2 Schritt 2: Von der Monade zum Monoid

Die von obigem *fold-left* verwendete Funktion ist nicht assoziativ, da sie die Monadenoperation *bind* benutzt. Der Operator *bind* hat den Typ $M\ \alpha \rightarrow \left(\alpha \rightarrow M\ \beta\right) \rightarrow M\ \beta$

und kann von daher nicht assoziativ sein. Dieses Problem läßt sich, wie bereits angesprochen dadurch lösen, daß man die Monade $(IO, unit, bind)$ zu einem Monoid $(IO', \oplus, \oplus_{id})$ umformt.

Das erreichen wir, indem wir den Parameter unseres Typkonstruktors *IO* entfernen, und die Funktionen *bind*, *unit* und *pprint* entsprechend anpassen. Man kann das in diesem Fall tun, da keine der Operationen einen anderen Wert als *()* zurückliefert. Wenn wir das Ergebnis einer Operation also ohnehin schon kennen, dann können wir auch auf seine Ausgabe verzichten:

```
type IO = String -> String
infixr 9 `bind`

unit :: IO
unit = \s -> s

bind :: IO -> IO -> IO
l `bind` r = r . l

pprint :: String -> IO
pprint str = \s -> s ++ str
```

Die Definitionen mußten, wie man sieht, gegenüber der vorangegangenen Version nur geringfügig modifiziert werden. Die eigentliche Schleife ist fast völlig unverändert geblieben[15]:

```
helloWorlds :: Int -> IO
helloWorlds n
  = for i <- [1..n] do
      pprint "Hello World " `bind`
      pprint (show i)       `bind`
      pprint "\n"
```

Und natürlich kann man diese *for*-Schleife jetzt wiederum mittels *fold-left* ausdrücken:

```
helloWorlds :: Int -> IO
helloWorlds n
  = foldl bind unit (map bodyFn [1..n])
  where
    bodyFn i
      = pprint "Hello World " `bind`
        pprint (show i)       `bind`
        pprint "\n"
```

Die von *fold* verwendete Funktion *bind* ist jetzt assoziativ, und *bind* und *unit* bilden ein Monoid. Also kann das erste Dualitätstheorem angewendet und *foldl* durch *foldPar* ersetzt werden. Das ist soweit richtig, allerdings kommt die Ernüchterung schnell, wenn man sich die Definitionen von *bind* und *unit* ansieht, und sich das *leaky fold-left law* ins Gedächtnis zurückruft: das oben stehende *fold-left* ist nichts weiter als eine Instanz des *leaky fold-left laws*. Offensichtlich ist eine weitere Transformation nötig.

### 5.2.3 Schritt 3: Das *leaky fold-left law* umgehen

Wir haben das Monoid $(M', bind, unit)$ und die Menge H der Funktionen, die vor dem Entfernen des Parameters aus unserem Typkonstruktor M ein Ergebnis vom Typ M () geliefert

[15] Zudem ist die Schleife besser lesbar geworden, da wir in der Definition *bind* als rechtsassoziativen Infixoperator der Priorität 9 definiert haben und somit auf die störenden Klammern verzichten können.

haben (im konkreten Beispiel ist $H = \{pprint\}$). Was wir jetzt benötigen ist eine Funktion $g :: \alpha \to M'$, die sämtliche Funktionen der Menge H modelliert. Mehrere Instanzen der Funktion g können dann durch Funktionskomposition mittels *bind* im Rumpf der *for*-Schleife verwendet werden. Um eine solche Schleife erfolgreich parallelisieren zu können muß die Funktion g im Schleifenrumpf die Struktur $\lambda\, s\ i \to s \oplus f\ i$ haben, damit das *fold-map fusion law* anwendbar bleibt. Wir definieren g demnach wie folgt:

$$g \quad v \ = \ \lambda\, s \ \to \quad s \oplus \quad f \quad v$$

$\oplus$ und f sind dabei entsprechend den Funktionen in der Menge H zu wählen. Verknüpft man zwei Instanzen von g mittels *bind*, so erhält man aufgrund der Assoziativität von $\oplus$ wiederum eine Instanz der Lambda-Funktion, die für eine Anwendung des *fold-map fusion laws* erforderlich ist. Auf diese Weise können beliebig viele Instanzen von g im Rumpf der Schleife verwendet werden:

$$\begin{array}{ll}
g\ v_1\ 'bind'\ g\ v_2 & \\
\Rightarrow g\ v_2 \circ g\ v_1 & \textit{Definition von bind} \\
\Rightarrow (\lambda\, s \to s \oplus f\ \ v_2) \circ (\lambda\, s \to s \oplus f\ \ v_1) & \textit{Definition von } g \\
\Rightarrow \lambda\, s \to (s \oplus f\ \ v_1) \oplus f\ \ v_2 & \textit{Definition von } \circ \\
\Rightarrow \lambda\, s \to s \oplus (f\ \ v_1 \oplus f\ \ v_2) & \textit{Assoziativität von } \oplus
\end{array}$$

Um das Monoid $(M', bind, unit)$ nun in seine endgültige Form zu bringen wird es mit Hilfe der Definition von g in das Monoid $(M'', \oplus, \oplus_{id})$ transformiert und die Funktion g wird durch die Funktion f ersetzt. Dieser Vorgang wird solange wiederholt, bis M'' entweder ein nicht-funktionaler Typ ist oder der Vorgang scheitert.

Nachfolgend wird nun diese Transformation an dem Beispiel der Monade für die Textausgabe durchgeführt. Wie bereits erwähnt besteht die Menge H hier nur aus einem Element, nämlich der Funktion *pprint*:

$$\begin{array}{c}
\text{pprint} \equiv g \\
\lambda\, str\ \ s \to s ++ str \equiv \lambda\, v\ \ s \to s \oplus f\ \ v \\
\textit{mit}\ \ x \oplus y \ = \ x ++ y \\
\textit{und}\ \ f\ \ v = v
\end{array}$$

Als neues Monoid haben wir somit $(String, ++, [])$ erhalten und *pprint* wird zur Identitätsfunktion. Damit ist die Umformung des Monoids abgeschlossen, denn mit String haben wir schließlich einen nicht-funktionalen Typen erhalten. Das Ergebnis sieht folgendermaßen aus:

```
type IO = String

unit :: IO
unit = []

bind :: IO -> IO -> IO
l `bind` r = l ++ r

pprint :: String -> IO
pprint str = str
```

Die eigentliche Schleife kann unverändert aus der letzten Version übernommen werden und wir haben jetzt eine Version, die sich erfolgreich parallelisieren läßt, d. h. wir können *foldl* gegen *foldPar* austauschen und der gewünschte Erfolg wird sich einstellen. Da die Transformationen des Monoids zum Erfolg geführt haben, handelte es sich bei der zu Beginn vorgestellten Monade um eine Vektorisierungs-Monade.

# 6 Zusammenfassung

Auf den zurückliegenden Seiten wurde dargestellt, wie man mit einigen wenigen Gesetzen imperative *for*-Schleifen mit Hilfe der Funktionen *fold* und *scan* ausdrücken kann, und unter welchen Bedingungen sich solche Schleifen parallelisieren lassen. Anstatt diese Aufgabe einem Compiler zu übertragen – was leicht fehlschlagen könnte – wurden die im ersten Teil gemachten Beobachtungen dazu verwendet, einen auf Monaden basierenden, allgemeinen Rahmen zu schaffen, in dem sich nur noch solche Schleifen ohne Verletzung der Monadengesetze und Typisierungsregeln schreiben lassen, die dann auch tatsächlich parallelisiert werden können. Die Struktur der eigentlichen Schleife bleibt bei den im Rahmen der Parallelisierung durchgeführten Transformationen fast unverändert, denn im wesentlichen werden nur die Monadenfunktionen *bind* und *unit* modifiziert. Die Idee besteht also weniger darin, vorhandene Schleifen in eine parallelisierbare Form zu transformieren, sondern vielmehr in der Beschreibung der Vektorisierungs-Monade als Programmiertechnik zum Erstellen von parallelisierbaren Programmen.

# 7 Literatur

[1] Richard Bird: „Algebraic identities for program calculation“, The Computer Journal, 32(2):122-126, 1989

[2] Richard Bird, Philip Wadler: „Introduction to Functional Programming“, Prentice Hall International Series in Computer Science, 1993

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: „Introduction to Algorithms“, MIT Press, 1990

[4] Jonathan M. D. Hill, Keith M. Clarke: „An introduction to category theory, category theory monads and their relationship to functional programming“, Technical Report 681, QMW CS, 1993

[5] Jonathan M. D. Hill, Keith M. Clarke: „Parallel Haskell: The vectorisation monad“, Technical Report 658, QMW CS, Dezember 1993

[6] Jonathan M. D. Hill, Keith M. Clarke, Richard Bornat: „The vectorisation monad“, Proceedings of PASCO´94, World Scientific Publishing Company, 1994

[7] Rishiyur. S. Nikhil, Arvind, James Hicks, u.a.: „pH Language Reference Manual, Version 1.0 - preliminary“, MIT Laboratory for Computer Science, 1995