Bedarfsgesteuerte Programmierung

Eine Einführung anhand von Haskell

Benedikt Grundmann – 8. April 2004

Zusammenfassung

Das sogenannte bedarfsgesteuerte Programmieren, im englischen unter dem griffigerem Namen “lazy programming” bekannt, hat sich als eine praktische und nützliche Programmiertechnik im Rahmen des funktionalen Programmierens erwiesen. Im folgenden wird ein Überblick über diese Technik und ihren Nutzen anhand der Programmiersprache Haskell gegeben. Besonderer Wert wird dabei auf die Darstellung von Techniken wie Monaden gelegt, die es trotz nicht vorhandener Seiteneffekte und nicht strikter Auswertung ermöglichen Algorithmen und Techniken aus der imperativen Programmierung im funktionalen Umfeld zu verwenden.

1 Geschichte von Haskell 1
2 Auswertungstechniken 2
2.1 Call-by-value 2
2.2 Call-by-name 3
2.3 Call-by-need 3
3 Strikte Funktionen 4
4 Strukturierung durch unendliche Datenstrukturen 4
5 Seiteneffekte 6
5.1 Referentielle Transparenz 8
5.2 Seiteneffekte emulieren: Zustand fädeln 8
5.3 Die Welt fädeln: E/A 9
5.4 Monade 10
5.4.1 Der `>>=` Operator 11
5.4.2 Syntaktischer Zucker `do` 11
6 Ausblick: Lazieness in strikten Programmiersprachen 12
6.1 Bedarfssteuerung als Bibliothek 12
6.2 Bedarfssteuerung als Spracherweiterung 13
7 Fazit 14
8 Quellen 14

1 Geschichte von Haskell

1985 entwickelte David Turner die erste bedarfsgesteuerte funktionale Programmiersprache Miranda. Innerhalb kurzer Zeit entwickelten sich an den Universitäten und

Forschungszentren sehr viele weitere funktionale Programmiersprachen und Dialekte. Als Reaktion darauf wurde Ende der 80er Jahre beschlossen eine standardisierte funktionale bedarfsgesteuerte Programmiersprache zu entwickeln. Die erste vollständige Definition dieser Sprache genannt Haskell nach dem berühmten Mathematiker Haskell Curry wurde 1990 veröffentlicht.

Von Anfang an war Haskell als Sprache konzepiert die aus einem stabilem Sprachkern und verschiedenen Erweiterungen besteht, die später nach und nach in den Kern aufgenommen werden können, falls sie sich als praktikabel erwiesen haben. Dies wurde auch nach der letzen Standardisierung 1999 als Haskell 98 durchgehalten und führte dazu daß Haskell immer noch eine der fortschrittlichsten funktionalen Programmiersprachen ist und wahrscheinlich noch sehr lange sein wird.

2 Auswertungstechniken

Um bedarfsgesteuerte Programmierung zu verstehen ist es sinnvoll sich als erstes die verschiedenen Techniken zur Auswertung von Ausdrücken anzusehen. Auswertungsregeln beschäftigen sich immer mit zwei Dingen:

- Wann und auf welche Weise Ausdrücke ausgewertet werden
- Wie und mit was dabei Bezeichner ersetzt werden

Im folgendem werde ich folgende Definition

```
x = 3 * 4
```

und diesen Ausdruck

```
5 * 5 + x + x
```

in denen Beispielauswertungsprotokollen verwenden.

2.1 Call-by-value

Die wahrscheinliche bekannteste Technik ist unter dem Namen “call-by-value” bekannt. Und wird in vielen bekannten Programmiersprachen wie SML, C, Pascal, ... verwendet.

- Bezeichner durch Werte ersetzt,
- der Ausdruck von innen nach aussen und
- von links nach rechts ausgewertet

$$
\begin{aligned}
5 * 5 + x + x &\rightarrow 5 * 5 + 12 + 12 \\
&\rightarrow 25 + 12 + 12 \\
&\rightarrow 37 + 12 \\
&\rightarrow 49
\end{aligned}
$$

Man kann die Auswertung solcher Ausdrücke auch als das reduzieren baumartiger Strukturen auf einen einzelne Knoten ansehen. Dies gilt auch bei der "call-by-name" Auswertungstechnik.

2.2 Call-by-name

Im Gegensatz zur call-by-value Auswertungsregel werden bei call-by-name die Bezeichner nicht durch Werte sondern durch den jeweils definierenden Ausdruck ersetzt.

$$
\begin{aligned}
5 * 5 + x + x &\rightarrow 5 * 5 + (3 * 4) + (3 * 4) \\
&\rightarrow 25 + (3 * 4) + (3 * 4) \\
&\rightarrow 25 + 12 + (3 * 4) \\
&\rightarrow 37 + (3 * 4) \\
&\rightarrow 37 + 12 \\
&\rightarrow 49
\end{aligned}
$$

Ganz offensichtlich gibt es bei dieser Technik einen imminenten Nachteil: Ausdrü cke werden unter Umständen mehr als einmal ausgewertet!

2.3 Call-by-need

Dieses Problem läßt sich jedoch sehr leicht beheben, wenn wir uns während der Auswertung bereits ausgewertete Ausdrücke merken. Diese optimierte Variante von call-by-name nennt man call-by-need und das ist die von Programmiersprachen wie Haskell verwendete Technik.

$$
\begin{aligned}
5 * 5 + x + x \text{ where } x = 3 * 4 &\rightarrow 25 + x + x \text{ where } x = 3 * 4 \\
&\rightarrow 25 + x + x \text{ where } x = 12 \\
&\rightarrow 37 + 12 \\
&\rightarrow 49
\end{aligned}
$$

Im Beispiel sollte auffallen dass der Bezeichner `x` nun nicht mehr zweimal mit gleichen Ausdrücken ersetzt wird sondern in beiden Fällen durch den *selben* Ausdruck. Somit ist die Datenstruktur auf die wir während der Auswertung arbeiten kein Baum mehr sondern ein azyklischer Graph. Tatsächlich ist es so das unser Baum auch ein Zyklen enthalten kann, da Ausdrücke ja erst ausgewertet werden, wenn sie

tatsächlich verwendet werden. Im **Abschnitt 4** wird noch genauer auf die Konsequenzen dieser Möglichkeit eingegangen. Die natürlich auch schon bei call-by-name zur Verfügung steht.

3 Strikte Funktionen

Was genau ist also mit dem Begriff *strikt* gemeint? Es ist ein Begriff zur Charakterisierung von dem Verhalten von Funktionen in Abhängigkeit ihrer Argumente. Informell gesprochen ist eine Funktion strikt in einem ihrer Argumente wenn sie divergiert sofern dieses Argument divergiert.

Definition

Für jede Funktion f gilt:

$$f \text{ist strikt} \Leftrightarrow f\bot = \bot$$

wobei $\bot$ (bottom) der Wert divergierender Ausdrücke ist.

In Haskell und anderen bedarfsgesteuerten Programmiersprachen gibt es auch Funktionen die nicht in allen ihren Argumenten *strikt* sind.

Zum Beispiel sind viele der in anderen Programmiersprachen eingebauten Sprachkonstrukte in Haskell durch Funktionen darstellbar. Als einfaches Beispiel seien hier die Funktionen `&&` und `||` genannt, die boolsches `AND` bzw. `OR` realisieren.

```
False && x = False
True  && x = x

False || x = x
True  || x = True

(1 + 2 == 3) || (1 ‘div‘ 0 == 0)  ===> True
```

4 Strukturierung durch unendliche Datenstrukturen

Wie bereits erwähnt ermöglicht die bedarfsgesteuerte Auswertung zyklische Ausdrücke zu schreiben. Mit anderen Worten erhält man somit unendliche Datenstrukturen. Eine besonders wichtige solche Datenstruktur ist der Strom, eine potentiell unendliche und bei Bedarf berechnetete Liste.

Das wohl trivialste Beispiel hierfür ist ein Strom gleicher Werte zum Beispiel ein Strom von Einsen.

```
ones :: [Int]
ones = 1 : ones
```

Sehr oft definiert man Haskell Funktionen welche Ströme erzeugen, hier wird der Strom der natürlichen Zahlen definiert:

```
enum :: Int -> [Int]
enum n = n : enum (n + 1)

nats :: [Int]
nats = enum 1
```

Folgendes Beispiel zeigt wie anhand von bedarfsgesteuerter Auswertung Algorithmen formuliert werden können, die so in strikten Sprachen nicht formuliert werden können.

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Hier wird in linearer Zeit ein Strom der Fibonacci Zahlen berechnet. Man sieht sehr deutlich das ohne Gedächniss, also bei call-by-need die Auswertung sehr ineffizient wäre.

Somit kann man anhand der Ströme Programme strukturieren und eine Aufteilung in Produzenten und Konsumenten erhalten. Beispiele für Produzenten wurden oben gezeigt, ein Konsument wäre zum Beispiel die Funktion `fib`, welche die `n`-te Fibonaccizahl berechnet.

```
fib :: Int -> Int
fib n = take (n - 1) fibs
```

Im allgemeinen kann man durch Ströme folgenden Programmaufbau erzielen:

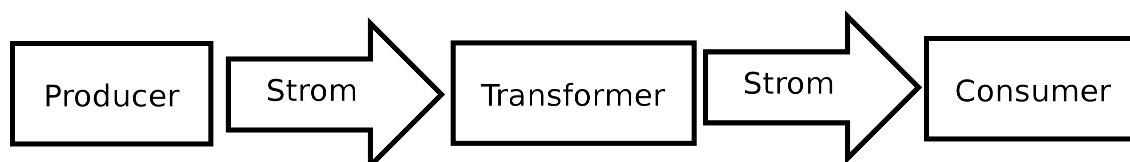


Figure 1 Programmstruktur

wobei der Transformatoren sowohl Verändern als auch Filtern.

Ein einfaches Beispiel für einen Produzenten *und* einen Transformatoren ist folgende Implementierung des Siebes des Eratosthenes:

```
primes       = sieve [2 .. ]
sieve (x:xs) = x : sieve [ y | y <- xs, y `mod` x > 0]
```

Was gewinnt man durch diese neuartige Strukturierungsmöglichkeit? Unabhängigkeit! Die Produzenten können geschrieben werden, ohne das man genau wissen muss wie sie später verwenden werden, oder wie viele Werte sie produzieren müssen.

Oftmals kann man bei Verwendung bedarfsgesteuerter Programmierung seinen Code so strukturieren, daß er zwar auf logisch sehr großen Datenstrukturen arbeitet, davon aber immer nur ein sehr kleiner Teil tatsächlich im Speicher existiert. Ein Beispiel hierfür sind viele Algorithmen der künstliche Intelligenz die bei Spielen verwendet wird. Hierbei betrachtet man im allgemeinen Teile des Baumes aller möglichen Züge von einem bestimmten Spielzustand aus. Da diese Bäume sehr groß werden, kann man sie nicht ohne weiteres direkt in strikten Sprachen modellieren. In Sprachen wie Haskell jedoch ist es sehr einfach. Gehen wir von der Existenz einer Funktion

```
moves :: GameState -> [GameState]

moves st = ...
```

die Spielzustände berechnet die von einem Zustand in einem Zug erreichbar sind. Mit deren Hilfe läßt sich oben erwähnter Baum wie folgt berechnen.

```
data Tree = Node GameState [Tree]

gameTree :: GameState -> Tree
gameTree startState = Node startState (map gameTree (moves startState))
```

Dieser Baum kann bei selbst bei einfachen Spielen wie TicTacToe sehr groß werden, dank bedarfsgesteuerter Auswertung wird aber erst bei tatsächlicher Verwendung eines Knotens diese auch berechnet. Bei vielen solchen Algorithem ist es außerdem nicht erforderlich die Spielstände weiterzuverwenden über die man eine bestimmte Spielposition erreicht hat (also weiter oben im Baum sind). Demnach kann der Garbage Collector der verwendeten Sprache den davon benutzen Speicher wiederverwenden und es wird immer nur Speicher führ sehr wenige Knoten tatsächlich benötigt.

5 Seiteneffekte

Leider hat diese zusätzliche Freiheit einen gewissen Preis! Man weiss nun nicht mehr ohne weiteres wann ein Ausdruck ausgewertet wird! Und das kann im Zusammenhang mit Seiteneffekten leider zu großen Problemen führen.

Ein Beispiel:

```
a = inputInt ()
b = inputInt ()
c = f a b
```

In diesem Beispiel gehe ich davon aus das es eine Funktion

```
inputInt :: () -> Int
```

gibt, welche eine Zahl einliest. Doch leider ist an dieser stelle überhaubt nicht klar, ob der Benutzer jemals überhaupt zwei Werte einzugeben hat, und wenn ja ob der erste eingegebene Wert an den Bezeichner `a` oder den Bezeichner `b` gebunden wird! Denn ob und wie oft (also einmal, zweimal oder gar keinmal) die Funktion `inputInt` aufgerufen wird hängt ausschliesslich von der Funktion `f` ab. Allerdings kann man sofern für die primitiven Funktionen bekannt ist ob und unter welchen Bedingen sie strikt in ihren Parametern sind, die Auswertungsreihenfolge immer bestimmen. Wäre zum Beispiel im obigen Beispiel an Stelle der unbekannten Funktion `f` die Funktion

```
(+) :: Int -> Int -> Int
```

verwendet worden, von der festgelegt ist das sie strikt in ihren Argumenten ist und zwar von links nach rechts, dann wäre klar das zuerst `a` und danach `b` an Werte gebunden worden wäre.

Mit anderen Worten die durch bedarfsgesteuerte Programmierung gewonnenen Strukturierungsmöglichkeiten führen zu Problemen sobald wir über die Reihenfolge der Auswertung von Ausdrücken Bescheid wissen müssen. Dies geschieht zwar nur im Zusammenhang mit Seiteneffekten, doch leider gibt es sehr viele verschiedene Arten von Seiteneffekten:

- Eingabe und Ausgabe
  - auf dem Terminal
  - in Dateien
  - bei graphischen Benutzeroberflächen
- Lesen und Schreiben von veränderlichen Variablen
- Hierbei besonders wichtig sind Reihungen (engl. Arrays)
- Ausnahmebehandlung

Dennoch hatten sich die Designer von Haskell dazu entschlossen, den einfachst möglichen Weg zu wählen obiges Problem zu umgehen:

In Haskell gibt es keine Möglichkeit Seiteneffekte auszudrücken

5.1 Referentielle Transparenz

Durch diesen Verzicht hat Haskell aber noch eine weitere interessante Eigenschaft erlangt. Die sogenannte referentielle Transparenz.

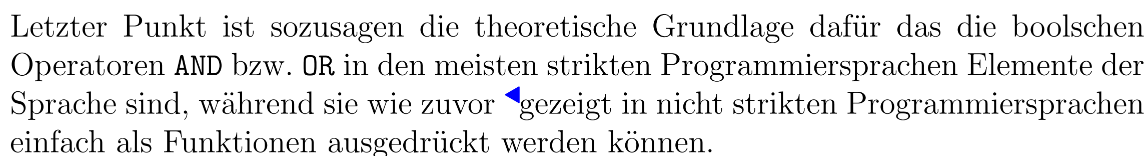
Definition

Sprachen ohne Seiteneffekte sind *referentiell transparent*, denn für sie gilt:

- alle Ausdrücke haben einen Wert (evtl. $\bot$[1]),
- ein Bezeichner kann immer durch den ihn definierenden Wert ersezt werden
- und der Wert eines Ausdruckes ist nur von den Werten seiner Teilausdrücke abhängig

In nicht strikten Programmiersprachen gilt außerdem, daß

- ein Bezeichner nicht nur durch den Wert sondern immer auch durch den jeweiligen Ausdruck ersetzt werden kann.
- Die Reihenfolge der Auswertung der Teilausdrücke ist irrelevant, da das Divergieren eines Teilausdruckes nicht notwendigerweise das Divergieren des gesamten Ausdruckes zur Folge hat.



Letzter Punkt ist sozusagen die theoretische Grundlage dafür das die boolschen Operatoren `AND` bzw. `OR` in den meisten strikten Programmiersprachen Elemente der Sprache sind, während sie wie zuvor gezeigt in nicht strikten Programmiersprachen einfach als Funktionen ausgedrückt werden können.

5.2 Seiteneffekte emulieren: Zustand fädeln

Dennoch kann natürlich keine Programmiersprache, die in der realen Welt eingesetzt werden soll, vollständig auf Seiteneffekte verzichten! Wie kann man also in einer reinen funktionalen Programmiersprache Seiteneffekte ausdrücken?

Die bekannteste und einfachste Technik hierfür ist das sogenannte “Durchfädeln von Zustand”. So sieht das klassische Beispiel der Fakultätsfunktion in einer imperativen Sprache (hier `C`) so aus:

[1] $\bot$ wurde eingeführt bei der **Striktheitsdefinition**

```
int fac (int n) {
    int i;
    int res = 1;
    for (i = 1; i <= n; i++)
      res *= i;
    return res;
}
```

In einer funktionalen Sprache kann man dasselbe erreichen indem man den jeweiligen Wert der Variable `res` durchreicht:

```
fac n = fac' 1 1
where
fac' i res = if i <= n then fac' (i + 1) (res * i)
            else res
```

5.3 Die Welt fädeln: E/A

Selbst Eingabe und Ausgabe läßt sich auf diese Weise modellieren! Denn Funktionen die Eingabe oder Ausgabe machen verändern die Welt. Also vorausgesetzt es gäbe so etwas wie einen Wert `world :: World` der den aktuellen Zustand der Welt darstellt, dann sähe ein entsprechendes Programm zum Beispiel so aus:

```
inputLine :: World -> (String, World)
print :: String -> World -> World

main :: World -> World
main world =
  let world'          = print "Enter your name" world
      (name, world'') = inputLine world'
  in print "Hello " ++ name world''
```

Durch das explizite “Herumreichen” der Welt ist auch das Problem der Auswertungsreihenfolge beseitigt. Da jede Funktion welche die Welt benutzt “automatisch” dafür sorgt, dass der vorherige diese Welt berechnende Ausdruck auch wirklich ausgeführt wird.

Wie kommt man an den Anfangswert der Welt? Man könnte sich folgendes vorstellen: Jedes Programm müßte eine Funktion

```
main :: World -> World
```

definieren, die von dem Laufzeitsystem mit der “Startwelt” aufgerufen wird.

Leider ist diese Model so ohne weiteres nicht einsetzbar! Man betrachte folgendes Programm:

```
main :: World -> World

main world =
  let _      = print "a" world
  in print "b" world
```

Was würde der Benutzer nun auf dem Bildschirm sehen? Ein einzelnes `a` oder ein einzelnes `b` oder gar `ab`. Nun vom Model her gäbe es nun gar nicht mehr einen sondern sogar zwei Benutzer! Denn es wurde eine Kopie der Welt angelegt! Über eine Programmiersprache mit derartigen Möglichkeiten nachzudenken ist sicherlich aus philosophischen Gründen her interessant aber wohl kaum praktikabel.

Wie kann man dieses Problem nun umgehen? Eine Möglichkeit wäre es Regeln in die Sprache einzubauen, die das Kopieren der Welt verhindern. Dies wurde tatsächlich in einer existierenden rein funktionalen Programmiersprache gemacht `CLEAN`. Dort wurde das Typesystem so erweitert, dass man Werte von denen keine Kopien angelegt werden dürfen ausdrücken kann.

In Haskell wurde jedoch eine andere Technik realisiert, die keine Änderungen am Typsystem erforderte[2]. Die sogenannten Monade.

5.4 Monade

Was ist also ein Monad? Ein Monad ist ein Typkonstruktor auf den die Funktionen `return` und `>>=` (bind) definiert sind. In Haskell wird das über folgende Typklasse[3] ausgedrückt.

```
class Monad m where
 (>>=)   :: m a -> (a -> m b) -> m b
 return  :: a -> m a
```

Informell ausgedrückt kann man mit jedem Monaden zwei Dinge tun:

- Mit Hilfe von `return` einen Wert in einen Monaden "stecken"
- Und mit Hilfe von `>>=` neue Monade aus bestehenden Monaden bauen.

Dabei beschreibt ein Monad `M a` im allgemeinen eine Abfolge von Aktionen, bei deren Ausführung am Ende ein Wert vom Typ `a` zurückgegeben wird. Wie ein Monad ausgeführt wird, ist dabei monad spezifisch. Es gibt also keine allgemeine Funktion `runMonad :: m a -> a` oder ähnliches.

Betrachten wir zuerst unser obiges Beispiel unter Verwendung von Monaden:

[2] stattdessen wurde viel syntaktischer Zucker eingeführt
[3] Lesen sie hierzu bitte auch die Einführung von Mark Kaminski

```
main :: IO ()

main =
  putStrLn "Enter your name" >>= \ () ->
  getLine >>= \name ->
  putStrLn ("Hello " ++ name)
```

`main` ist hier also keine Prozedur mehr sondern ein Monad. Genauer gesagt ein Monad vom Typ `IO ()`. Analog zu obigen Überlegungen werden tatsächlich vom Haskelllaufzeitsystem `IO`-Monade die im Toplevel eingegeben werden sofort ausgeführt.

5.4.1 Der `>>=` Operator

Um obiges Beispiel zu verstehen ist es sehr wichtig den `>>=` Operator verstanden zu haben.

```
(>>=) :: m a -> (a -> m b) -> m b
```

Er erhält zwei Argumente einen Monaden `m` und eine Funktion `f`. Daraus baut `>>=` einen neuen Monaden, welcher wenn er ausgeführt wird folgendes macht:

1. `m` ausführen und mit dem resultierenden Wert `w`

2. die Funktion `f` aufrufen.

Dabei kann es durchaus Monade geben bei denen die Funktion `f` niemals aufgerufen wird. So ist zum Beispiel der Datentyp `Maybe` auch ein Monad mit dem man Berechnungen darstellen kann, die fehlschlagen können oder optional sind.

```
data Maybe a = Just a | Nothing

return = Just

Nothing >>= f = Nothing
Just v  >>= f = f v
```

5.4.2 Syntaktischer Zucker `do`

Es ist sehr einfach aus bestehenden Monaden neue zu bauen. Oben bereits gesehener Monad `getLine` kann wie folgt definiert werden:

```
getLine :: IO String
```

```
getLine =
  getChar >>=                                      -- lese ein Zeichen
  \c -> if c == '\n'                                -- wenn neue Zeile
          then return ""                            -- dann leeren String
          else                                      -- sonst das zeichen und
            getLine >>= \l -> return (c : l)       -- rest der Zeile
```

Dennoch wird der Code durch die vielen notwendigen Abstraktionen leicht unübersichtlich. Daher wurde in Haskell syntaktischer Zucker für `>>=` eingeführt. So könnte man `getLine` auch wie folgt schreiben:

```
getLine :: IO String

getLine =
  do c <- getChar              -- lese ein Zeichen
     if c == '\n'              -- wenn Neue Zeile
       then return ""          -- dann leeren String
       else do l <- getLine    -- ansonsten rekursiv
               return (c : l)  -- zeichen und rest der Zeile
```

`do` wird dabei vom Compiler folgendermaßen übersetzt:

```
do v1 <- action1
   v2 <- action2
   monad v1 v2
```

wird expandiert zu

```
action1 >>= \v1 -> action2 >>= \v2 -> monad v1 v2
```

6 Ausblick: Lazieness in strikten Programmiersprachen

Es wurde gezeigt was für Vorteile bedarfsgesteuerte Programmierung bietet und wie man einer bedarfsgesteuerten Programmiersprache Elemente aus der strikten Programmierung wieder hinzufügen kann. Es stellt sich natürlich die Frage ob man nicht auch den anderen Weg gehen kann. Ist es möglich in einer strikten Programmiersprache bedarfsgesteuerte Auswertung einzuführen?

Es ist prinzipiell auf zwei verschiedenen Weisen möglich. Zum einen kann man die Sprache weitestgehend unverändert lassen und die bedarfsgesteuerte Auswertung über die bestehenden Sprachmittel einführen.

6.1 Bedarfssteuerung als Bibliothek

Dazu wird für gewöhnlich ein extra Typ für bedarfsgesteuerte Ausdrücke eingeführt, der intern den auszuwertenden Ausdruck über eine Closure oder ähnliches darstellt.

Ein Beispiel für eine derartige Unterstützung für bedarfsgesteuerte Programmierung ist in OCAML realisiert worden:

```
# let n = lazy (factorial 12);;
val n : int lazy_t = <lazy>
# Lazy.force n + 1;;
- : int = 479001601
```

Wobei in OCAML syntaktischer Zucker für die Erstellung bedarfsgesteuerter Ausdrücke Teil der Sprache ist.
Eine mögliche Implementierung[4] sähe zum Beispiel folgendermaßen aus:

```
type 'a status =
  | Delayed of (unit -> 'a)
  | Value of 'a

type 'a t = 'a status ref

let force l =
  match !l with
  | Value v   -> v
  | Delayed f -> let v = f () in l := Value v; v

let _lazy f = ref (Delayed f)
```

Wobei der Compiler offensichtlich das `lazy` Konstrukt folgendermaßen übersetzt:

```
lazy (3 + 4) ==> _lazy (fun () -> 3 + 4)
```

Leider ist dieses Konstrukt völlig orthogonal zu allen anderen Sprachelementen und führt somit durch den eigenen Typ bedarfsgesteuerter Ausdrücke dazu, daß alle Funktionen für bedarfsgesteuerte Ausdrücke neugeschrieben werden müssen!

6.2 Bedarfssteuerung als Spracherweiterung

Dieses Problem kann nur umgangen werden, indem man bedarfsgesteuerte Ausdrücke in die Sprachsemantik aufnimmt. Dies wurde zum Beispiel in der Programmiersprache Alice getan. Hier sind noch nicht ausgewertete Ausdrücke nicht Teil des statischen Typsystems sondern analog zu $\perp$ besondere Werte die jeder Ausdruck haben kann.

```
- val n = lazy (factorial 12);
val n : int = _lazy
```

[4] Die ursprüngliche Implementierung in OCAML war sehr ähnlich.

```
- n + 1;
val it : int = 479001601
```

Somit besteht das zuvor im Abschnitt **6.1** gezeigte Problem nicht mehr und man kann bestehende Funktion mit bedarfsgesteuerten Ausdrücken verwenden.

Dennoch ergeben sich in der Praxis öfters Probleme, da in der bedarfsgesteuerten und in der strikten funktionalen Programmierung unterschiedliche Methoden üblich und effizient sind. Aus diesem Grund ist es doch sehr oft sinvoll verschiedene Funktionen zu definieren.

Betrachten wir hier als Beispiel die bekannten Funktionen zur Faltung einer Liste. In strikten Programmiersprachen verwendet man für gewöhnlich `foldl`, da diese Funktion *endrekursiv* also iterativ programmiert werden kann.

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs
```

Ein offensichtlicher Nachteil ist jedoch das `foldl` erst dann einen Wert zurückgeben kann, wenn die gesamte Liste bearbeitet worden ist! Somit ist `foldl` nicht geeignet um unendliche Listen zu bearbeiten oder als praktikabler Transformator zu agieren.

Dementsprechend wird in bedarfsgesteuerten Programmiersprachen `foldr` verwendet, denn diese Funktion ist *nicht* strikt in der Liste sofern die übergebene Prozedur *nicht* strikt im zweiten Argument ist.

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

7 Fazit

Es wurde gezeigt das bedarfsgesteuerte Programmierung eine praktische und interessante Technik ist, die sich in vielen Bereichen sinnvoll einsetzen läßt. Es ist dennoch immer noch nicht klar ob es einen besten Weg gibt bedarfsgesteuerte Programmierung zu unterstützen. Wie dargelegt wurde hat sowohl standardmäßig bedarfsgesteuerte Auswertung als auch standardmäßig strikte Auswertung eine Reihe von Vorteilen und Nachteilen. Nichts destotrotz ist es auf jeden Fall lohnenswert sich detailierter mit dieser Technik auseinanderzusetzen.

8 Quellen

1. J.Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98-107,1989.

2. S. P. Jones, J. Hughes, et al. *Report on the Programming Language Haskell 98*, February 1999.

3. R.Plasmeijer and M. van Ekelen. *Clean Version 2.0 Language Report*, December 2001. Draft

4. S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, New York;Amsterdam;Bonn, 1999

5. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannad, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of LNCS, pages 1-16, Nancy, France, September 1985. Springer

6. P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993