"Lazy evaluation"

Vor- und Nachteile gegenüber Parameterübergabe per Wert und per Referenz

Arne Steinmetz - mi3055

12.05.2002

Inhalt

"Lazy evaluation"

[Inhalt]...[Nicht-strikte Auswertung]

1. Einleitung

Das Informatik-Seminar der Fachhochschule Wedel im Sommersemester 2002 befasst sich unter anderem mit dem Thema "Funktionale Programmierung in Haskell" (betreut von Prof. Dr. Uwe Schmidt). Diese Ausarbeitung beschreibt die sogenannte "Lazy evaluation", sowie deren Vor- und Nachteile gegenüber "Call-by-value" und "Call-by-reference".

1.1 Die Sprache Haskell

Funktionale Sprachen bestehen wie der Name sagt ausschließlich aus Funktionen. Ein in funktionaler Sprache verfasstes Programm ist ebenfalls eine Funktion. Die Sprache Haskell ist eine "pur funktionale", "nicht-strikte" Programmiersprache. Haskell an sich ist nicht Thema dieser Ausarbeitung, syntaktische und semantische Komponenten werden nur in so weit erläutert, wie sie zum Verständnis des Prinzips der "Lazy evaluation" beitragen. Hinter diesem Begriff steckt das in Haskell realisierte Konzept der "nicht-strikten Auswertung" von Ausdrücken. Für das allgemeine Verständnis soll im Vorfeld jedoch das Typkonzept und Funktionen in Haskell unter die Lupe genommen werden.

1.2 Funktionen als Gleichungen

Entsprechend des Haskell-Typmodells können Teilausdrücke zu Funktionsaufrufen ausgewertet werden. Ein Funktionsaufruf besteht aus einem konventionsgemäßen Namen sowie gegebenenfalls einer Liste von aktuellen Parametern. Funktionen werden in Haskell entweder in Form einer (optionalen) Typsignatur sowie einer Gleichung bzw. einer Folge von Gleichungen deklariert, oder in Form einer sogenannten "Lambda-Abstraktion".

Code-Beispiel in Haskell [Quelldatei]

```
module HaskFuncs where

-- Beispiel fuer Funktion
  mult :: Float -> Float -> Float
  mult a b = a * b

-- 2. Beispiel fuer Funktion
  mult2 :: Float -> (Float -> Float)
  mult2 a b = a * b
```

Das obige Code-Beispiel stellt ein Haskell-Modul mit Namen "HaskFuncs" dar. Es kann beispielsweise mit dem HUGS98-Compiler aus einer Ascii-Datei geladen werden. Innerhalb des Moduls werden zwei Funktionen deklariert, "mult" und "mult2". Sie stehen nach Laden des Moduls zur Verfügung und können an der Kommando-Zeile ausgeführt werden.

Die jeweils erste Zeile stellt die Deklaration der Typsignatur dar, und damit eine explizite Typbindung für die Funktion. "A Gentle Introduction to Haskell" empfiehlt die Verwendung von Typsignaturen, weil sie die Lesbarkeit des Codes erhöhen, und die Fehlererkennung vereinfachen. Der Doppelpunkt steht für "hat den Typ", während "->" als Zuordnungs- oder Funktionspfeil bezeichnet wird. Beide Funktionen sind Ausdrücke vom Typ **Float** und akzeptieren als Parameter genau zwei **Float**-Ausdrücke. Keine der Funktionen darf mit anderen Parametern als genau zwei **Float**-Ausdrücken aufgerufen werden, andernfalls wird zur Compile-Zeit ein Fehler generiert. Die zweite Zeile stellt die Gleichung und damit den eigentlichen Funktionsausdruck dar. Im Beispiel werden hier jeweils die beiden formellen Parameter der Funktion multipliziert. Bei erfolgreicher Typprüfung wird die Funktion zum Wert ihrer Gleichung ausgewertet. Schlägt die Typprüfung fehl, meldet der Compiler einen Fehler.

Funktionsaufruf (HUGS98)

```
HaskFuncs>mult 3 2
6.0
HaskFuncs>mult2 (-1) 2
-2.0
HaskFuncs>
```

Ausgabe des HUGS98 bei fehlgeschlagener Typprüfung

```
HaskFuncs>mult ’a’ 2
ERROR - Type error in application
*** Expression : mult ’a’ 2
*** Term : ’a’
*** Type : Char
*** Does not match : Float

HaskFuncs>
```

Durch Haskells statisches Typsystem wird der Fehler im Funktionsaufruf zur Compilezeit erkannt. Der Compiler meldet einen Typfehler in der Anwendung. Der erste Parameter mit dem Wert ’a’ wird geprüft, und ein "Missmatch" zwischen den Typen **Char** und **Float** wird festgestellt. Für die Übergabe von komplexen Ausdrücken als Parameter ist die Klammerung Pflicht, da sonst Parserfehler entstehen. Beispielsweise führt der Aufruf "`mult -1 1`" zu einem Compiler-Fehler, weil bei der Auswertung von links nach rechts das "-" als Zeichen behandelt wird. Deshalb: "`mult (-1) 1`".

1.3 Rechts-Assoziativität der Typbindung

Der Zuordnungspfeil "->" ist rechts-assoziativ, so dass die Typsignaturen von "mult" und "mult2" äquivalent sind.

Rechts-Assoziativität der Typbindung			
mult ::	**Float** ->	**Float** ->	**Float**
	multipliziere Wert von b	*Ordne Ausdruck Wert von a zu*	*Neuer Ausdruck*
mult2 ::	**Float** ->	(**Float**->**Float**)	
	multipliziere Wert von b	*Speichere Wert von a als neuen Ausdruck*	

Äquivalenz der Typsignaturen

1.4 Funktionsanwendung

Die Anwendung einer Funktion ist links-assoziativ, die Funktion wird von links nach rechts auf die Parameter angewendet. Somit werden die aktuellen Parameter gemäß der Reihenfolge der formellen in die Funktionsgleichung eingesetzt.

Die entspricht der Äquivalenz mathematischer Funktionen bei links-assoziativer Zuordnung:

Gegeben seien die algebraischen Funktionen f und g mit
`f:x->z ;`
`g:x->y ;`
Dann gilt:
`h = f(g(x)) <=> h:(x->y)->z <=> h:x->y->z ;`

Praktische Anwendung von Assoziativität und Äquivalenz (HUGS98)

```
HaskFuncs> mult2 1 2
2.0
HaskFuncs> (mult2 1) 2
2.0
HaskFuncs>
```

Die Typen dieser beiden Ausdrücke **Float**->**Float**->**Float** und **Float**->(**Float**->**Float**) sind wegen der Rechts-Assoziativität der Typsignaturen äquivalent. Die Auswertung erfolgt dabei wie bei arithmetischen Funktionen von innen nach außen:
`=> h=f(g(x)) <=> h:(x->y)->z`

"Curried Functions"

Eine Funktion mit f mit n Parametern kann mit Hilfe einer "Curried Function" auch als eine Funktion g mit einem Parameter angegeben werden, welche eine Funktion mit n-1 Parametern zurückliefert.
Die oben aufgelisteten Beispiel-Funktionen "mult" und "mult2" sind "Curried Functions".
In abstrakter Darstellung lautet ein Aufruf der Funktion "mult2":

`mult2 FloatExp1 FloatExp2`, wobei "FloatExp1" und "FloatExp2" Ausdrücke vom Typ **Float** seien.

Aufgrund der Links-Assoziativität der Funktionsanwendung ist folgender Aufruf äquivalent:

```
(mult2 FloatExp1) FloatExp2
```

Die Bezeichnung rührt vom Namen des vermeintlichen Entdeckers dieser Funktionen her, "Curry Haskell", nach auch die Programmiersprache benannt wurde. Praktisch kann mit "Curried Functions" kann das Prinzip der "partiellen Anwendung" von Funktionen realisiert werden.

"Partielle Funktionsanwendung"

Gegeben sei eine Funktion in Haskell, die einen numerischen Ausdruck halbiert.

```
half a = / a 2.0
```

Die Funktion ist beschrieben durch die Division mit den Parametern `a` und `2.0`. Mit einer entsprechenden "Curried Function" ließe sie sich folglich als Funktion mit einem Parameter darstellen. Zur Verfügung stehen uns die dazu die "mult" oder "mult2"-Funktion.

```
half = mult 0.5
```

Diese Definition ist ein Beispiel für "partielle Funktionsanwendung". Außerdem wird eine Funktion als Wert des Ausdrucks zurückgeliefert.

[Inhalt]...[Nicht-strikte Auswertung]

"Lazy evaluation"

2. Nicht-strikte Auswertung

2.1 "Lazy evaluation" oder "Call-by-need"

In "Haskell" werden Ausdrücke grundsätzlich nicht strikt ausgewertet. Ein (Teil-)Ausdruck wird erst durch seinen Wert ersetzt, wenn dieser zum Beispiel für einen arithmetischen Vergleich benötigt wird. Demzufolge wird auch ein als aktueller Parameter einer Funktion übergebener Ausdruck erst ausgewertet, wenn sein Wert innerhalb des Funktionsrumpfes verwendet wird. Wurde der Wert einmal berechnet, werden alle namentlichen Vorkommen des Ausdrucks durch seinen Wert ersetzt. Ein einfaches Beispiel verdeutlicht diese Strategie.

```
>> const1 a = 1
```

Definiert ist eine konstante Funktion "`const1`" mit einem beliebigen Parameter "`a`" und dem Rückgabewert 1. Der Rückgabewert folgender beispielhafter Funktionsaufrufe ist also immer 1.

(Pseudocode)
```
>> const1 10 => 1
>> const1 (1+4) => 1
>> const1 'A' => 1
>> const1 (1/0) => 1
```

Für das Ergebnis der Funktion ist der Wert des aktuellen Parameters irrelevant. Er wird von einem Interpreter deshalb nie ausgewertet. Aus diesem Grund führt der letzte Aufruf mit der Division durch Null als Parameter nicht zu einem Fehler.

Solange also der Wert eines Ausdrucks (einer Funktion) nicht benötigt wird, behandelt "Haskell" diesen als eine Definition. Als solche werden auch entsprechende Teilausdrücke behandelt.

2.2 Hintergrund

Definierte Teilausdrücke werden innerhalb von Ausdrücken oder als aktuelle Parameter zunächst nur durch einen Namen identifiziert. Aufgrund der Verwendung der Namen in Operationen muss der Interpreter entscheiden, ob der Wert des Ausdrucks für das Ergebnis der Operation relevant ist. Zu diesem Zweck werden definierte Ausdrücke auf Basis des Lambda-Kalküls zerlegt. Die resultierenden "Unbekannten" sind Bezeichner für Teilausdrücke (Funktionen). Das folgende erweiterte Beispiel zeigt, dass die Analyse des Umfeldes von Ausdrücken recht komplex werden kann.

2.3 Ein Anwendungsbeispiel

Als Beispiel wird einfache dreidimensionaler Vektortyp definiert. Für die Verarbeitung werden Instanzen der Klasse "Eq" für den Gleichheitstest und der Klasse "Show" für die String-Ausgabe an der Kommandozeile erzeugt.

Code-Beispiel in Haskell [Quelldatei]

```
>> data LVect3D = LVect3f {u,v,w::Float}
>> newvect u v w = LVect3f u v w
>> instance Show LVect3f
>>     where show n = show(u n)++" "++show(v n)++" "++show(w n)
>> instance Eq LVect3f
>>     where (==) a b = ((u a)==(u b)&&(v a)==(v b)&&(w a)==(w b))
>> (newvect (1+2) 2 4)==(newvect 2 (2/0) 4) => False
>> (newvect 1 2 4)==(newvect 1 (2/0) 4) => _|_
```

Die beiden Gleichheitstest verdeutlichen die Strategie, wegen derer die nicht-strikte Auswertung auch als "Call-by-need" bezeichnet wird. Die aktuellen Parameter sind nach Erzeugung eines neuen Vektors mit der "`newvect`"-Methode zunächst Ausdrücke vom Type **`Float`** mit den Namen u,v und w. Der Vergleich der Vektoren gegeneinander basiert auf dem arithmetischen Gleichheitstest der einzelnen Koordinaten, also der Ausdrücke u,v,w. Um diesen Test durchzuführen, müssen die Teilausdrücke ausgewertet werden. Das Ergebnis des Tests ist durch die Und-Verknüpfung der Einzelvergleiche definiert, deren Teilausdrücke zu `True` ausgewertet werden müssen, um als Ergebnis `True` zu erhalten.

Beim ersten Aufruf wird diese Bedingung bereits bei Auswertung des ganz linken Ausdrucks verletzt. Zunächst wird die Addition ausgewertet, ihr Ergebnis mit dem Wert 2 verglichen. Das Ergebnis ist `False`, womit der gesamte Test fehlgeschlagen ist. Die restlichen Ausdrücke werden nicht berechnet.

Im zweiten Beispiel liefert der erste Gleichheitstest `1==1 => True`. Folglich ist der zweite Teilausdruck relevant für den Gesamtausdruck. Hier führt jedoch die Auswertung des arithmetischen Ausdrucks `2/0` zu einem Laufzeitfehler, da das Ergebnis nicht definiert ist (_|_).

2.4 Unendliche Strukturen

Mit Hilfe nicht-strikter Auswertung kann eine interessante Speicherstruktur realisiert werden, eine unendliche Liste. Eine solche Liste kann in "Haskell" rekursiv erzeugt werden. Da funktionale Programme nicht linear abgearbeitet werden, und mit Hilfe der "Lazy Evaluation", kann es dabei nie zu einer nicht terminierenden Rekursion kommen. Folgende Funktion erzeugt eine unendliche Liste aus einem **Integer**-Anfangsausdruck, in dem sie zu dem jeweils eingefügten aktuellen Ausdruck den Folgewert hinzufügt.

Code-Beispiel in Haskell [Quelldatei]

```
>> -- unendliche liste

>> intlist :: Integer -> [Integer]
>> intlist i = i : intlist (i+1)
```

Aus der Erfahrung mit imperativen Programmiersprachen würde man vermuten, dass diese Funktion nie terminiert. Dass dem nicht so ist, zeigen die beispielhaften Aufrufe:

```
>> zehnints = take 10 (intlist 0)
>> von13bis21 = take 9 (intlist 13)
```

(Pseudocode)

```
>> zehnints => [0,1,2,3,4,5,6,7,8,9]
>> von13bis21 => [13,14,15,16,17,18,19,20,21]
```

Durch "Lazy Evaluation" werden nie alle Elemente der Liste berechnet. Ausgewertet werden nur die (rekursiven) Aufrufe von "`intlist`", welche zur Berechnung des geforderten Listenausschnitts benötigt werden (die Funktion "`take x`" liefert die ersten x Elemente einer Liste zurück).

[Inhalt]...[Einleitung]...[Strikte Auswertung]

"Lazy evaluation"

[Inhalt]...[Nicht-strike Auswertung]...["Strictness" in Haskell]

---

3. Strikte Auswertung

3.1 Erklärung

Die meisten Programmiersprachen, insbesondere die imperativen und Objekt-orientierten, werten Ausdrücke strikt aus. In diesem Fall wird ein Ausdruck nie als solcher behandelt, sonder stets zu einem Wert ausgewertet. Daraus resultiert die Tatsache, dass eine Funktion, die auf einem nicht terminierenden Ausdruck arbeitet, selbst nicht terminiert.

Als Beispiel sei eine Funktion gegeben, die zwei ganze Zahlen durcheinander teilt, welche als Parameter übergeben werden. Ist der Divisor (im Beispiel der zweite Parameter) null, wird die Berechnung der Division innerhalb der Funktion einen Fehler auslösen. Das Programm wird, falls keine Fehlerbehandlung besteht, sofort beendet. Die Funktion terminiert nicht, und liefert keinen Wert zurück.

- **Beispiel in Ansi C**

  Der Code:

```
#include <stdio.h>

int idiv (int a, int b)
{
  return a/b;
}

int main (void)
{
  fprintf(stdout,"2 / 3 = %d\n", idiv(2,3));
  fprintf(stdout,"2 / 0 = %d\n", idiv(2,0));
  return 0;
}
```

  Ausgabe des Programms (gcc unter cygwin32):

```
2 / 3 = 0
0 [main] fdiv 1628 handle_exceptions: Exception:
STATUS_INTEGER_DIVIDE_BY_ZERO
[...]
```

Aufgrund der strikten Auswertung wird von der Funktion "idiv" nicht der Ausdruck `a/b` zurückgeliefert, sondern sein Wert. Im ersten Fall wird `2/3` zu null ausgewertet, da "drei null mal in zwei passt". Im zweiten Fall muss `2/0` berechnet werden, was zu einem arithmetischen Ausnahmefehler führt, weil der Wert nicht definiert ist. Die Auswertung bricht ab, ebenfalls die Funktion, sowie das gesamte Programm. Ein Rückgabewert ist nicht definiert. Der "Haskell 98 Report" sieht folgende Notation vor: Der Wert eines nicht terminierenden Ausdrucks wird mit "_|_" (sprich engl. "bottom") angegeben.

Die strikte Auswertung spielt insbesondere bei der Parameterübergabe per Wert ("Call-by-value", wie oben) und per Referenz ("Call-by-reference") eine Rolle, welche im folgenden noch einmal vorgestellt werden.

3.2 "Call-by-value"

- Deutsche Bezeichnung: Übergabe per Wert, Wertparameter

Definition "Call-by-value"
Übergabeart für Parameter einer Prozedur, bei der beim Prozeduraufruf nur der Wert des aktuellen Parameters übergeben wird, nicht jedoch der Name oder die Adresse, unter welcher der Ausdruck im Speicher steht. Quelle: "Duden Informatik", 2. Auflage, Dudenverlag 1993

- **Erläuterung**
  Bei der Übergabe per Wert werden innerhalb einer aufgerufenen Funktion private Kopien der übergebenen Parameter angelegt, auf denen die Funktion arbeitet. Entsprechend der im Funktionskopf angegebenen formellen Parameter wird bei Aufruf eine Kopie jedes aktuellen Parameters im Stackbereich der Funktion gespeichert. Verdeutlicht wird dies durch die Ausgabe folgenden Programms. Es gibt zunächst die Adresse einer Variable der Hauptprozedur aus, welche dann als Parameter an die Funktion "padress" übergeben wird. Diese gibt erneut die Adresse des aktuellen Parameters aus.

  Beispiel für kopierte aktuelle Parameter in Ansi C:

```
#include <stdio.h>

void padress (int i)
{
  printf("Funktion: &i = %p\n",&i);
}

int main (void)
{
  int i;
  printf("Hauptprogramm: &i = %p\n",&i);
  padress(i);
  return 0;
```

```
}
```

Die Ausgabe des Programms

```
Hauptprogramm: &i = 0x240feb0
Funktion: &i = 0x240fe8c
```

Der Wert der Variablen i hat also eine andere Speicheradresse, als der des Wertparameters i, was durch das Kopieren der aktuellen Parameter erklärt ist. Daraus folgt auch, dass keine Seiteneffekte auftreten können, falls der Wert des Parameters in der Funktion geändert wird. Nach Rückkehr in die Hauptprozedur kann auf dem ursprünglichen Wert weitergearbeitet werden, wie das zweite Beispiel zeigt.

Beispiel in C: [Quelldatei]

```
#include <stdio.h>

void tausche(int a,int b)
{
  int tmp=a;
  a=b;
  b=tmp;
}
int main (void)
{
  int x=1,y=2;
  fprintf(stdout,"Vor tausche(x,y) : x=%d y=%d\n",x,y);
  tausche(x,y);
  fprintf(stdout,"Nach tausche(x,y): x=%d y=%d\n",x,y);
  return 0;
}
```

Ausgabe des Programms:

```
Vor tausche(x,y) : x=1 y=2
Nach tausche(x,y): x=1 y=2
```

Die Funktion "tausche" operiert lediglich auf den kopierten aktuellen Parametern. Nach Beenden der Funktion besteht kein Zugriff mehr auf ihren Speicherbereich. Die in der Hauptprozedur gespeicherten Variablen x und y behalten den ursprünglichen Wert, somit erfüllt die Funktion nicht, was ihr Name verspricht...

3.3 "Call-by-reference"

- Deutsche Bezeichnung: Übergabe per Referenz, Referenzparameter

Definition "Call-by-reference"
Gängige Parameterübergabe in imperativen Programmiersprachen, wobei die Prozedur(en) unmittelbar mit den aktuellen Parametern arbeiten, und nicht nur auf Kopien der Werte. Quelle: "Duden Informatik", 2. Auflage, Dudenverlag 1993

Beispiel in C [Quelldatei]

```
#include <stdio.h>

void tausche(int * a,int * b)
{
  int tmp=*a;
  *a=*b;
  *b=tmp;
}
int main (void)
{
...
  tausche(&x,&y);
...
}
```

Ausgabe des Programms:

```
Vor tausche(x,y) : x=1 y=2
Nach tausche(x,y): x=2 y=1
```

- **Erläuterung**
  Der formale Parameter einer Funktion wird als Zeiger auf den Parametertyp deklariert. Bei Auswertung des aktuellen Parameters wird die Adresse des Ergebnisses verwendet, die bei Verwendung des formalen Parameters innerhalb der Funktion wird jeweils eingesetzt wird. Die Konsequenz ist, dass der aktuelle Parameter verändert wird, falls durch die Referenz Änderungen vorgenommen werden. Dieser Seiteneffekt ist bei richtiger Benutzung gewollt, z.B. beim Aufbau einer Liste.

  Hinweis:
  Formal unterstützt die Sprache C nur Wertparameter. Entsprechend des oben genannten Prinzips können Referenzparameter aber simuliert werden, in dem mit Zeigertypen gearbeitet wird.

[Inhalt]...[Nicht-strike Auswertung]...["Strictness" in Haskell]

"Lazy evaluation"

4. "Strictness" in Haskell

4.1 Erzwungene Auswertung

Die Auswertung von Teilausdrücken muss aufgrund der "Lazy evaluation" in "Haskell" unter Umständen erzwungen werden ("Force strictness"). Da standardmäßig nicht strikt ausgewertet wird, steht hierfür in "Haskell" die Funktion "`seq`" zur Verfügung.

```
>> seq :: a->(b->b)
```

Die Funktion erwartet als Parameter einen Ausdruck a und einen Funktionsausdruck (b->b), der auf a angewendet werden soll. Laut dem "Haskell 98 Report" ist ihr Wert folgendermaßen definiert:

```
>> seq _|_ f = _|_
>> seq x f = f x
```
(Pseudocode)

Das Zeichen "_|_" steht für einen nicht terminierenden Ausdruck (wie die Division durch null). Ist also der Ausdruck, auf den die Funktion angewendet werden soll, nicht definiert, so ist auch das Ergebnis der "`seq`" -Funktion undefiniert. Andernfalls ist das Ergebnis abhängig von dem Ausdruck (b->b). Die "`seq`"-Funktion kann als normale Funktion `seq x (f x)` oder als Infix-Operator `x ‘seq‘ f x` benutzt werden. Bei zusammengesetzten Datentypen muss dabei beachtet werden, dass "`seq`" nur die jeweils erste Ebene eines Ausdrucks (also z.B. bei einer Liste den Kopf) auswertet.
Für geschachtelte Ausdrücke kann zudem der Alias "`$!`" verwendet werden:

```
>> $! :: (a->b) -> a -> b
>> f $! x = x ’seq’ f x
```

4.2 Simulation von "Call-by-value"

Mit Hilfe der "`seq`"-Methode kann in "Haskell" das Prinzip von "Call-by-value" simuliert werden. Zu Demonstrationszwecken soll noch einmal das Code-Beispiel aus Kapitel 2 "Nicht-strikte Auswertung" herangezogen werden. Ein neuer Typ mit einer veränderten Konstruktor-Funktion, welche die aktuellen Parameter strikt auswertet, soll implementiert werden.

Code-Beispiel in Haskell [Quelldatei]

```
>> data SVect3D = SVect3f {u,v,w::Float}
>> newsvect x y z = ((SVect3f $! x) $! y) $! z
```

Die "Eq"-/ "Show"-Instanzen bleiben unverändert.

```
>> instance Show SVect3f
>>     where show n = show(u n)++" "++show(v n)++" "++show(w n)
>> instance Eq SVect3f
>>     where (==) a b = ((u a)==(u b)&&(v a)==(v b)&&(w a)==(w b))
```

Durch die Verwendung des "`$!`"-Operators werden bei Erzeugen eines neuen Ausdrucks vom Typ "`SVect3f`" alle drei übergebenen Float-Audrücke ausgewertet. Ein Vergleich, wie er in Kapitel 2 mit "Lazy Evaluation" ausgewertet wurde, schlägt nun fehl:

```
>> (newsvect (1+2) 2 4)==(newsvect 2 (2/0) 4) => _|_
```

Dieser Gleichheitstest führt zu einem Laufzeitfehler, weil der Ausdruck (2/0) bei Konstruktion des (zweiten) Vektors strikt ausgewertet wird.

4.3 "Strictness flags"

Das Beispiel für die Simulation von "Call-by-value" zeigt einen typischen Fall, in dem eine strikte Auswertung von Teilausdrücken gewünscht ist. Es kann sinnvoll sein, sicherzustellen, dass bei Konstruktion eines neuen Ausdruck eines bestimmten Typs alle Teilausdrücke terminieren (definiert sind). Für diesen Sonderfall existiert in "Haskell" ein weiteres syntaktisches Element, mit dem Parameter von Typ-Konstruktoren als strikt gekennzeichnet werden. Ein Ausrufezeichen vor der Typangabe des formellen Parameters im Konstruktor erfüllt diesen Zweck. Mit Hilfe dieser "strictness flags" kann der "Call-by-value"-Konstruktor abgekürzt implementiert werden. Dazu noch einmal das Beispiel mit dem Vektor-Typ:

Code-Beispiel in Haskell [Quelldatei]

```
>> data SFVect3D = SFVect3f !Float !Float !Float
>> newsfvect f g h = SFVect3f f g h
```

Das Verhalten entspricht dem "Call-by-value"-Beispiel.

[Inhalt]...[Strikte Auswertung]...[Vor-/Nachteile]

5. Vor-/Nachteile

5.1 "Handling"

In einer nicht-strikten, funktionalen Sprache ist die Reihenfolge der Auswertung von Ausdrücken nicht relevant. Somit muss sich auch der Entwickler/Programmierer hierüber keine Gedanken machen. Des weiteren kann er sich darauf verlassen, dass alle (im Zweifel rechenintensiven) Operationen nur dann berechnet werden, wenn ihr Ergebnis für die Weiterverarbeitung benötigt wird. Dadurch wird in vielen Fällen ein problemnaher Entwurf begünstigt, der strukturelle Aspekte der Implementation kaum berücksichtigen muss.
Es besteht beispielsweise die Möglichkeit, Verarbeitungs- oder Kontrollroutinen für Datenstrukturen zu implementieren, die eventuell nicht terminieren. Aufgrund von Parametern oder Ausprägungen der Daten kann die Auswertung dann gesteuert werden.

5.2 Komplexität

Die Komplexität eines Algorithmus oder einer Funktion bezeichnet die Wachstumsrate von Resourcen wie zum Beispiel der Laufzeit gegenüber der Menge der zu verabeitenden Daten. Im Falle von "Lazy evaluation" wird immer die geringste Anzahl an benötigten Werten von Ausdrücken berechnet. Daraus lässt sich schließen, dass in speziellen Problemfällen die Komplexität von Algorithmen in einer nicht strikten Sprache reduziert werden kann. Als Fallbeispiel soll an dieser Stelle der einfache "Insertion-Sort"-Algorithmus zum Sortieren von Feldern dienen.

Code-Beispiel in Haskell [Quelldatei]

```
>> insert :: Integer->[Integer]->[Integer]
>> insert a [] = [a]
>> insert a (x:xs) | a<x = [a]++(x:xs)
>>     | otherwise = [x]++(insert a xs)

>> sort :: [Integer]->[Integer]
>> sort [] = []
>> sort (x:xs) = (insert x (sort xs))
```

An zwei Stellen spart die nicht-strikte Auswertung in Einzelfällen Laufzeit ein, weil die Anzahl der auszuwertenden Teilausdrücke reduziert wird. In Kapitel 2 "Nicht-strikte Auswertung" wurde darauf hingewiesen, dass alle namentlichen Vorkommen eines ausgewerteten Ausdrucks durch seinen Wert ersetzt werden. Damit reduziert sich bei doppelten Elementen oder wiederholten Vergleichstests im zu sortierenden Feld die Menge der auszuwertenden Ausdrücke. Das gleiche gilt für den rekursiven Aufruf der "`insert`"-Methode. Wiederholte Einfügeoperationen werden nur einmal ausgewertet.

Während also die Komplexität des "Insertion-Sort"-Algorithmus in imperativen oder objektorientierten Programmiersprachen als quadratisch zur Anzahl der sortierenden Elemente bekannt ist, hängt sie bei einer nicht-strikten, funktionalen Sprache wie "Haskell" vor allem auch von der Ausprägung des Feldes ab.

5.3 Speicherbedarf

Durch Reduktion der Auswertungen werden Operationen gespart, und damit Laufzeit und Speicher. Zwei Aspekte müssen bei der Programmierung mit nicht-strikter Auswertung jedoch bedacht werden, so dass sich dieser Vorteil nicht umkehrt.

Der erste stellt einen Fall dar, der auch in der "Haskell Mailing List" mehrfach Thema war. Beim Einlesen von Dateien generieren Parser häufig Hilfsstrukturen, in denen die Inhalte der Dateien gepackt werden. Aus diesen Strukturen werden schließlich die relevanten Daten extrahiert. Wenn in diesem Fall die von einem Parser erzeugten Strukturen nicht strikt ausgewertet werden, kann beim Einlesen vieler großer Strukturen der Speicherbedarf erheblich ansteigen.

Der zweite Aspekt besteht darin, dass für die Verwaltung nicht strikter Ausdrücke zusätzlicher Speicher benötigt wird, um die Zuordnung von namentlichen Definitionen zu Ausdrücken zu realisieren, und eine Veranschlagung der auszuwertenden und nicht auszuwertenden Teilausdrücke vorzunehmen, sowie Informationen.

[Inhalt]...["Strictness" in Haskell]...[Literaturverzeichnis]

"Lazy evaluation"

[Inhalt]...[Vor-/Nachteile]

---

6. Literaturverzeichnis

6.1 Bücher

- "Duden Informatik", 2. Auflage, Dudenverlag 1993

6.2 WWW-Links

- "A Gentle Introduction To Haskell", Hudak, Peterson, Fasel: http://haskell.org/tutorial/
- "Haskell 98 Report", verschiedene: http://haskell.org/
- "Beginning Haskell", David Mertz: http://www-105.ibm.com/developerworks/
- "Why functional programming matters", John Hughes: http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html

---

[Inhalt]...[Vor-/Nachteile]

"Lazy evaluation"

[Inhalt]

Downloads

Begleitmaterial Vortrag

- HTML-Version
- GNUZip-Tar-Archiv der HTML-Dateien

Dieses Dokument

- PDF-Version
- GNUZip-Tar-Archiv der HTML-Dateien

[Inhalt]

Funktionale Programmierung - Lazy evaluation

Table of Contents