Programmiersprachen im Multicore-Zeitalter: Haskell

Colin Benner

30. Januar 2012

1 Einführung

Mit der ständig wachsenden Anzahl verfügbarer Prozessor-Kerne wird das Schreiben paralleler Programme immer wichtiger. Häufig werden dazu Mechanismen wie *pthreads* oder *OpenMP* eingesetzt, die eine hohe Performance-Ausbeute ermöglichen, im Gegenzug aber viel Arbeit bei der Synchronisation erfordern. Grundsätzlich erfordert die parallele Programmierung mit imperativen Sprachen Kontrolle beim Zugriff auf gemeinsamen Speicher. Dies ist fehlerträchtig und verursacht in vielen Fällen Fehler, deren Ursache schwer auszumachen ist, etwa im Falle von race conditions.

Der Grund dafür ist, dass in imperativen Programmiersprachen Nebeneffekte etwas Selbstverständliches sind und das Ändern vorhandener Daten der Normalfall ist. Ganz anders ist die Herangehensweise bei funktionalen Programmiersprachen. Hier ist es der Normalfall, das Funktionen keine Nebeneffekte haben und Datenstrukturen nur einmal geschrieben und anschließend nicht mehr verändert werden. Statt die Daten zu ändern wird eine geänderte Version an einer anderen Stelle im Speicher abgelegt und die alte Version bleibt erhalten, bis der Garbage Collector sie, wenn sie nicht mehr genutzt werden, freigibt.

Dadurch hat man es beim Schreiben paralleler Programme mit einer funktionalen Programmiersprache in der Regel deutlich einfacher. Um eine explizite Synchronisierung muss sich der Programmierer höchstens bei einem kleinen imperativen Teil des Programms kümmern, beim Rest übernimmt der Compiler beziehungsweise die Laufzeitumgebung diese Aufgabe.

Besonders einfach verhält sich die Sache bei einer rein funktionalen Programmiersprache wie Haskell, bei der imperative Konstrukte mit Hilfe des ausdrucksstarken Typsystems auf rein funktionale abgebildet werden. Dadurch ist es verhältnismäßig leicht möglich Programme zu schreiben, die bei der parallelen Ausführung grundsätzlich immer das selbe Ergebnis, wie bei der sequentiellen Ausführung liefern. Das einzige, was sich verändert ist die Laufzeit des Programms.

2 Grundlagen von Haskell

Bei Haskell handelt es sich um eine statisch typisierte rein funktionale Programmiersprache mit call-by-need-Semantik, die mit verschiedenen Compilern in hoch optimierten Maschinencode übersetzt werden kann, der bei entsprechender Programmierung durchaus mit der Performance von in C geschriebenen Programmen mithalten kann.

Wie auch in ML nimmt der Compiler dem Programmierer mit Hilfe von Typinferenz viel Arbeit ab, erlaubt aber ebenfalls explizite Typannotationen, die in der Regel für den Leser des Codes hilfreiche Informationen bereithalten und häufig als ein wesentlicher Teil der Dokumentation betrachtet werden. Dadurch, dass das Typsystem mächtiger als das von ML ist, sind allerdings an manchen Stellen explizite Typannotationen erforderlich, da das Problem der Typinferenz von Haskell-Programmen nur in den „einfachen“ Fällen entscheidbar ist.

2.1 Syntax

Haskell verwendet eine ML-inspirierte Syntax mit guter Unterstützung für pattern matching, die im Folgenden sehr kurz beschrieben werden soll.

Der Code zum Filtern einer Liste nach Elementen, die einem bestimmten Prädikat genügen, sieht etwa folgendermaßen aus:

```
1 filterList :: (a -> Bool) -> [a] -> [a]
```

deklariert filterList als eine Funktion, die als erstes Argument eine Funktion, die ein Element vom Typ a auf ein Element des Booleschen Typ Bool abbildet und als zweites Argument eine Liste von as erhält und eine Liste von as zurückliefert. Inspiriert ist diese Form der Notation durch die in der Theorie der Programmierung verwendete Notation *Bezeichner* :: *Typ*. Groß geschriebene Bezeichner sind Typ- oder Konstruktornamen, kleingeschriebene Variablen oder Typvariablen. Wenn eine nicht gebundene Typvariable a in einem Typ typ auftritt, stet typ für $\forall$ a. typ (in Haskell forall a. typ).

Mit

```
2 filterList p [] = []
```

wird der Rekursionsanker definiert: Wenn das zweite Argument die leere Liste ist, so ist das Ergebnis die leere Liste.

```
3 filterList p (x:xs) | p x       = x : filterList p xs
4                     | otherwise = filterList p xs
```

In den Zeilen 3 und 4 wird definiert, was passiert, wenn das zweite Argument eine Liste ist, deren erstes Element x und deren Rest xs ist. Falls p x wahr ist, so ist das Ergebnis die Liste, die mit x beginnt und deren Rest filterList p xs ist, andernfalls wird x verworfen und nur filterList p xs zurückgegeben.

Das folgende Programm definiert die Funktion isPrime n, die angibt, ob n eine Primzahl ist und die Variable primes, die eine eine Liste aller Primzahlen enthält.

```
1 isPrime n = foldr (\p r -> p*p>n || (n`mod`p /= 0 && r)) True primes
2 primes = 2 : [ n | n <- [3,5..], isPrime n ]
```

Der in der ersten Zeile befindliche Operator (/=) ist der Test auf Ungleichheit, foldr entspricht im Wesentlichen MLs fold_right. Hier wird schlicht die Primalität durch Probedivision durch alle Primzahlen $\leq \sqrt{n}$ implementiert. \ x -> e ist dabei Haskells Schreibweise für die λ-Abstraktion $\lambda x.e$.

In Zeile zwei wird eine Liste mit Hilfe von (:) erzeugt, deren erstes Element 2 ist und deren Rest durch eine sogenannte *List Comprehension* beschrieben wird.

Diese an die Mengen-Notation aus der Mathematik angelehnte Schreibweise steht für Folgendes: Es wird eine Liste beschreiben, deren Elemente n der unendlichen Liste [3,5..] der ungerade Zahlen > 1 entnommen werden, wobei in der Ergebnisliste die nur Elemente n auftauchen, die gemäß isPrime prim sind.

Enstprechend könnte man auf der linken Seite der List Comprehension einen beliebigen Ausdruck, etwa 2*n-3 hinschreiben oder weitere Bedingungen für n hinzufügen, wie zum Beispiel n > 10 oder gcd n 7 /= 0:

```
1 foo = [ 2*n-3 | n <- [3,5..], isPrime n, n > 10, gcd n 7 /= 0 ]
```

Damit lässt sich die filterList-Funktion auch folgendermaßen schreiben:

```
1 filterList p lst = [ x | x <- lst, p x ]
```

was aber nur syntaktischer für

```
1 filterList p lst = map (\x -> x) (filter p lst)
```

ist, wobei filter eine eingebaute Funktion ist, die man ohne weiteres statt der hier definierten Funktion filterList einsetzen kann.

2.2 Einige Eigenschaften

Die Reihenfolge, in der die Funktionsdefinitionen im Quelltext auftauchen spielt bei Haskell keine Rolle.

Dadurch das Haskell rein funktional ist, muss der Programmierer die Ausführungsreihenfolge nicht explizit angeben und der Compiler kann diese nach belieben verändern. Das einzige, was Teile der Reihenfolge fest vorgibt, sind die Datenabhängigkeiten. Das macht es möglich, dass in Haskell verschiedene Optimierungen stattfinden, die in anderen Sprachen nicht oder nur in wenigen Spezialfällen möglich wären, wie die sogenannte *deforestation*, die es Haskell-Compilern erlaubt map f (map g list) in map (f . g) list zu transformieren, wobei . die der Funktionskompositionsoperator ist.

2.3 Auswertungsstrategie

Bei call-by-value wird bei einem Funktionsaufruf foo a zunächst das Argument a vollständig ausgewertet und jedes Vorkommen des formalen Parameters im Funktionsrumpf durch den Wert von a ersetzt. Wenn der formale Parameter im Funktionsrumpf nicht auftaucht war die Berechnung von a unnötig.

Bei call-by-name wird im Gegensatz dazu jedes Vorkommen des formalen Parameters im Funktionsrumpf durch den Ausdruck a ersetzt. Bei der Auswertung von foo a wird

also möglicherweise `a` mehrfach ausgewertet. In einer rein funktionalen Sprache kommt aber bei jeder dieser Auswertungen das selbe Ergebnis heraus.

Haskell verwendet eine Mischform: *call-by-need*. Dabei muss `a` nur ausgewertet werden, wenn das Ergebnis tatsächlich benötigt wird, wie bei call-by-name. Wird das Ergebnis einmal ausgerechnet steht es aber im Gegensatz zu call-by-name an allen anderen Stellen, an denen der formale Parameter in der Funktion auftritt ebenfalls (ohne nochmalige Auswertung von `a`) zur Verfügung.

Semantisch macht dies aber, da Haskell rein funktional ist keinen Unterschied, ob das Argument an mehreren Stellen oder nur an einer ausgewertet wird.

Implementiert ist call-by-need in Haskell als *Lazy Evaluation*, dass heißt der Wert eines Parameters wird erst berechnet, wenn er zum ersten Mal benötigt wird.

Dadurch müssen nur tatsächlich benötigte Ausdrücke überhaupt, aber keine mehrfach ausgewertet werden. Das macht es beispielsweise möglich mit unendlichen Datenstrukturen und Funktionen die mit call-by-value nicht terminieren würden sinnvoll zu arbeiten.

3 Parallele Programmierung mit Haskell

Die Ausführungsstrategie ähnelt dem Konzept der *Futures*, deren Ursprung in der asynchronen Interprozesskommunikation liegt. Ein Future ist ein Platzhalter, der für das Ergebnis einer (komplizierten) Berechnung steht. Mit dem Future kann dann weiter gerechnet werden, bis der eigentliche Inhalt tatsächlich benötigt wird. Erst zu diesem Zeitpunkt muss die Berechnung des eigentlichen Werts abgeschlossen sein. Dadurch wird automatisch eine Ausführungsreihenfolge gewählt, bei der alle Datenabhängigkeiten erfüllt sind und es ist somit möglich die Ausführung ohne großen Aufwand zu parallelisieren.

Bei Haskells Lazy Evaluation wird für jede Berechnung zunächst ein sogenannter *Thunk* erzeugt, der speichert, was eigentlich berechnet werden soll. Ausgewertet werden diese Thunks erst, wenn ihr Ergebnis benötigt wird. Auch hier sind also die Datenabhängigkeiten kodiert und man könnte Thunks die sowieso benötigt werden in einem parallelen Programm einfach ausrechnen, sobald dafür Rechenleistung verfügbar ist.

Während Haskells Nebeneffektfreiheit gewisse Fehler von Anfang an ausschließt, gibt es auch neue Fehlerquellen. Durch Lazy Evaluation ist es nicht ganz einfach die Ausführungsreihenfolge zu kontrollieren und es kann vorkommen, dass eine Berechnung, die von einem Thread im Hintergrund ausgeführt werden sollte tatsächlich vom Hauptthread erledigt wird.

Haskell bietet in der im Haskell98-Standard festgehaltenen Variante noch keine Unterstützung für Parallelisierung. Diese wird erst durch verschiedene Spracherweiterungen nachgerüstet. Die drei wichtigsten sollen im Folgenden genauer vorgestellt werden.

Nicht jede Haskell-Implementierung bietet jedoch diese Erweiterungen an, weshalb ich mich im von jetzt an auf die Implementierung durch den Glasgow Haskell Compiler (GHC) beschränken werde.

Um mit GHC ein Programm zu übersetzen, dass diese Erweiterungen mit Hilfe mehrerer Prozessorkerne zu nutzen, muss man den sonst üblichen Aufruf

```
ghc foo.hs
```

zum Übersetzen des Moduls foo, das sich in Datei foo.hs befindet folgendermaßen ändern:

```
ghc -threaded foo.hs
```

um mit einer SMP-fähige Laufzeitumgebung zu Linken, beziehungsweise bei neueren GHC-Versionen

```
ghc -threaded -rtsopts foo.h
```

damit man bei der Ausführung des Programms anbieten kann, wie viele Betriesbssystemthreads verwendet werden sollen.

Anschließend kann man das parallele Programm mittels

```
./foo +RTS -Nn
```

ausführen.

3.1 Glasgow parallel Haskell

Die zur Parallelisierung vielleicht wichtigste Haskell-Erweiterung ist *Glasgow parallel Haskell* (GpH), die dazu dient Programme durch Annotationen, die angeben, was parallel ausgeführt werden soll, zu parallelisieren.

Verwendet wird diese Erweiterung mit Hilfe von nur zwei Funktionen

```
1 par :: a -> b -> b
2 pseq :: a -> b -> b
```

aus dem Module Control.Parallel, die es ermöglichen die parallele Ausführung eines Programms zu erreichen und allgemein die Ausführungsreihenfolge zu bestimmen.

Dabei dient pseq a b schlicht dazu, zuerst a auszuwerten, bevor b zurückgegeben wird. Es gibt auch eine ähnliche, vom Haskell-Standard vorgeschriebene Funktion seq, bei der aber der Compiler zu Optimierungszwecken die Auswertungsreihenfolge der beiden Argumente vertauschen kann. Um erst a und dann b auszuführen ist diese Funktion also (zumindest bei Verwendung eines optimierenden Compilers wie GHC) nur bedingt sinnvoll.

Parallelismus kommt durch Verwendung von par zum Einsatz. Mit par a b teilt man dem Compiler mit, dass ein anderer Thread a auswerten und b zurückgeben soll. Dazu wird für a ein sogenannter *spark* (Funke) erzeugt, der ähnlich einem Thunk die durchzuführende Berechnung beschreibt. Anders als Thunks wird ein spark aber nicht erst dann ausgewertet, wenn er benötigt wird. Stattdessen werden die erzeugten sparks nacheinander von einem Thread im Hintergrund ausgewertet. Ein nicht mehr benötigter spark kann allerdings auch vorher vom Garbage Collector entfernt werden. Dadurch wird erreicht, dass (solange noch sparks vorhanden sind) keiner der verfügbaren CPU-Kerne untätig sein muss.

Beispiel 1. Die Verwendung von GpH zur Parallelisierung des folgenden Programms zur Berechnung der n-ten Fibonacci-Zahl

```
1 fib :: Int -> Int
2 fib 0 = 0
3 fib 1 = 1
4 fib n = fib (n - 1) + fib (n - 2)
```

lässt sich mit Hilfe von par und pseq folgendermaßen parallelisieren:

```
1 pfib :: Int -> Int
2 pfib 0 = 0
3 pfib 1 = 1
4 pfib n = a `par` (b `pseq` a + b)
5   where a = fib (n - 1)
6         b = fib (n - 2)
```

So werden die Berechnungen von fib (n-1) und fib (n-2) parallel ausgeführt und danach die Summe berechnet.

Ließe man hier pseq weg, also

```
4 pfib n = a `par` a + b
```

so würde a für die Berechnung im Hintergrund vermerkt, aber je nach Reihenfolge der Auswertung der Operanden von + möglicherweise sofort benötigt, so dass der Hauptthread zuerst auf das Ergebnis der Berechnung von a wartet und danach selbst b ausrechnet. Von einem gewissen Overhead abgesehen würde die Laufzeit also gegenüber dem sequentiellen Programm nicht verändert.

3.1.1 weak head normal form

In anderen Fällen (wo das Ergebnis der parallelen Berechnung komplizierter ist) geht dies nicht so leicht, etwa wenn a eine Liste liefert. In dem Fall würde von

```
1 foo = a `par` b
2   where a = map (+5) [1..1000000]
```

map (+5) [1..1000000] nicht vollständig ausgerechnet, sondern nur zu (1+5) : map (+5) [2..1000000] ausgewertet, die sogenannte *weak head normal form* (WHNF) des Ausdrucks berechnet. Das ist eine Form, in der der Ausdruck so weit wie möglich ausgewertet ist, ohne dass eine Auswertung innerhalb im inneren eines Konstruktors stattfände. Hier sorgt Haskells Lazy Evaluation-Strategie dafür, dass weniger ausgewertet wird, als man vielleicht intuitiv erwarten würde. Wenn nun in b das vollständige Ergebnis von a benötigt wird, so muss der Thread, der b ausführt auch den größten Teil der Auswertung von a übernehmen.

Um in einem solchen Fall parallel zur Berechnung von b den Ausdruck map (+5) [1..1000000] komplett auszuführen, muss man dafür sorgen, dass irgendwie die komplette Liste durchlaufen wird.

Eine Möglichkeit dafür ist die Verwendung der Funktion

```
1 forceList :: [a] -> ()
2 forceList [] = ()
3 forceList (x:xs) = x `pseq` forceList xs
```

die jedes einzelne Listenelement auf weak head normal form reduziert. Haben die Elemente der Liste einen der Basistypen, sind also nicht von der Form Konstruktor parameter1 parameter2 ... parameterN, so ist nach Ausführung von forceList lst die Liste lst vollständig ausgewertet, andernfalls muss man dafür eine Funktion wie forceList für den Elementtyp auf jedes einzelne Element anwenden.

Dazu gibt es den einen allgemeinen Mechanismus: Die Typklasse NFData definiert eine Funktion

```
1 rnf :: NFData a => a -> ()
```

die ihr Argument auf Normalform, dass heißt das, anders als bei der WHNF, auch das Innere eines Konstruktor ausgewertet wird, bringt. Da die Implementierung von rnf vom Typ a abhängt, muss dies Funktion für alle verwendeten Typen, deren Auswertung man erzwingen möchte, definiert werden. Für Listen könnte man rnf beispielsweise so definieren:

```
1 rnf = forceList
```

Für die wichtigsten eingebaute Typen ist rnf bereits definiert.

Für diese gibt es auch vorgefertigte Auswertungsstrategien, etwa rseq, die eine Auswertung zu WHNF bewirkt, rdeepseq, die eine vollständige Auswertung mittels rnf bewirkt, oder rpar, die eine parallele Auswertung startet.

Diese Strategien lassen sich auch zu neuen Strategien zusammensetzen.

Weiterhin gibt es Funktionen, die Strategien auf jedes einzelne Element eines Datentyps anwenden, etwa

```
1 parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

die eine Funktion auf alle Elemente einer Liste anwendet und das Ergebnis anschließend elementweise gemäß der angegebenen Strategie auswertet.

Durch die Haskells Nebeneffektfreiheit ist es semantisch egal, ob solche Berechnungen im Hintergrund ausgeführt werden, oder ob die Auswertung tatsächlich erst erfolgt, wenn das Ergebnis benötigt ist. Hat man also bereits ein korrektes sequentielles Programm, so kann man daraus mit par und pseq ein paralleles Programm machen, ohne dass man sich Sorgen machen müsste,

3.2 Concurrent Haskell

Bei *Concurrent Haskell* handelt es sich um eine Erweiterung, die in erster Linie dazu gedacht ist konzeptionell nebenläufige Programme in natürlicher Weise zu implementieren, etwa ein Server, der aufwendige Berechnungen durchführt während er im Hintergrund neue Anfragen entgegennimmt oder Daten von einem Speichermedium liegt. Concurrent Haskell bietet dazu die Möglichkeit explizit Userlevel- oder Betriebssystem-Threads zu erzeugen, die auf verschiedene Arten miteinander kommunizieren können.

Dadurch ist ist zwar grundsätzlich möglich, beliebige parallele Programme mit Concurrent Haskell zu implementieren, womit man sich aber teilweise (in vielen Fällen unnötigerweise) wieder die Probleme aus der Welt der imperativen Programmiersprachen einhandelte. Man sollte daher Concurrent Haskell nicht als Ersatz von GpH verwenden.

Zur Kommunikation gibt es sogenannte MVars („mutable variables“), darauf basierende *Kanäle* und *Software Transactional Memory* (STM).

Beispiel 2 (aus [1])**.**

```
1 import Control.Concurrent
2 import System.Directory
3 f = forkIO (writeFile "foo" "Hello world!") >> doesFileExist "foo"
```

gibt in einem neuen Thread „Hello world!“ in die Datei „foo“ aus und der Hauptthread anschließend zurück, ob die Datei existiert.

3.2.1 Kommunikation und Synchronisation mit MVars

MVars sind eine Art Container, in der sich ein Element eines bestimmten Typs befinden kann, aber nicht muss. MVar a ist der Typ eines Containers für Elemente des Typs a. Es gibt Funktionen zum Schreiben eines Elements in diesen Container und zum Auslesen eines Elements (sowohl mit, als auch ohne Entfernen aus dem Container). Die schreibenden Funktionen blockieren solange noch eine Element in der MVar ist, die lesenden solange kein Element enthalten ist. Mit Hilfe dieses Blockierens können MVars auch zur Implementierung von Synchronisation zwischen verschiedenen Threads eingesetzt werden.

Diese Operationen sind atomar, beim Schreiben in die MVar kann es also nicht zu Inkonsistenzen kommen. Race Conditions können wie bei nachrichtenbasierten Systemen also nur dadurch auftreten, dass Daten in der falschen Reihenfolge gelesen bzw. geschrieben werden.

Wichtige Funktionen sind etwa

Funktion	Beschreibung
newEmptyMVar	erzeugt einen neuen MVar-Container, der kein Element enthält
newMVar a	erzeugt eine neue MVar mit Inhalt a
takeMVar	entnimmt einer MVar ihren Inhalt
putMVar	legt ein Element im Container ab
readMVar	liest Inhalt des Container, ohne ihn zu leeren

3.2.2 Kanäle

Weiterhin gibt es die Möglichkeit, dass ein Thread einem anderen durch einen Kanal Daten schickt. Die durch den Kanal geschickten Daten müssen alle den selben Typ haben. Das Senden der Daten ist immer erfolgreich. Wenn sie nicht direkt gelesen werden werden sie einfach gepuffert. Das Lesen (und Entfernen) von Daten aus dem Kanal durch die Gegenseite ist hingegen eine blockierende Operation, die nötigenfalls wartet, bis Daten in den Kanal geschrieben werden.

Auch hierbei muss der Programmierer nur sicherstellen, dass die Reihenfolge der Schreib- und Leseoperationen stimmt.

Beispiel 3 (ebenfalls aus [1])**.**

```
1 import Control.Concurrent
2 import Control.Concurrent.Chan
3
4 chanExample = do
5   ch <- newChan
6   forkIO (do {writeChan ch "hello world"; writeChan ch "now i quit"})
7   readChan ch >>= print
8   readChan ch >>= print
```

erzeugt einen Kanal, in den der eine Thread zwei Nachrichten schreibt, die der andere anschließend liest und ausgibt.

3.2.3 Kommunikation mit STM

Statt der einfachen Datenübertragung in Form einzelner Elemente (MVars) oder von Listen von Elementen (Kanäle) gibt es die Möglichkeit, mit Hilfe von Software Transactional Memory komplexere Kommunikationsvorgänge auf sogenannten TVars auszuführen. Dabei legt jeder Thread, der auf eine TVar zugreift ein Log an, welche Daten er gelesen hat und welche er schreiben möchte. Vor dem tatsächlichen schreiben überprüft er dann, ob die gelesenen Daten mit den aktuellen übereinstimmen. Wenn ja, so schreibt er sein Ergebnis. Andernfalls hat ein anderer Prozess die Daten zwischenzeitlich bearbeitet und das Schreiben des schon berechneten Ergebnisses könnte zu einem inkonsistenten System führen. Daher startet der Thread die atomare Berechnung erneut, basierend auf den neuen Daten.

3.3 Data Parallel Haskell

Eine noch stark in der Entwicklung befindliche Erweiterung ist *Data Parallel Haskell*, womit man in Haskell sehr leicht vorhandene Datenparallelität ausnutzen kann.

Ungewöhnlich an Data Parallel Haskell ist die Tatsache, dass hier nicht nur sogenannter *flacher* Datenparallelismus, sondern auch verschachtelten Datenparallelismus („nested data parallelism“) ausgenutzt werden kann. Es gibt zwar verschiedene Programmiersprachen, die flachen Datenparallelismus automatisch ausnutzen können, wie etwa High Performance Fortran oder Compute Unified Device Architecture (CUDA), verschachtelten Datenparallelismus unterstützen diese jedoch nicht. Eine erste Implementierung von verschachteltem Datenparallelismus wurde 1993 in Form der Programmiersprache *NESL* veröffentlicht. Diese war jedoch ein reiner proof-of-concept. Die Sprache bot nur eine kleine Anzahl von fest eingebauten Typen, keinerlei Unterstützung für Funktionen höherer Ordnung und wurde noch dazu durch einen Interpreter implementiert.

Data Parallel Haskell basiert auf der Grundidee von NESL, geschachtelten Datenparallelismus zunächst in flachen Datenparallelismus zu transformieren und diesen dann mit den schon lange bekannten, einfachen Techniken auszunutzen.

Data Parallel Haskell bietet unter anderem einen parallelen Arraytyp mit der üblichen List Comprehension-Schreibweise

```
squareOdds lst = [: x^2 | x <- lst, isOdd x :]
```

die syntaktischer Zucker für

```
squareOdds lst = mapP (^2) (filterP isOdd lst)
```

ist. Dabei sind mapP und filterP datenparallele Varianten von map und filter.

Auch von vielen anderen Funktionen bietet Data Parallel Haskell im Module Data.Array.Parallel eine parallele Variante an:

Funktion	Beschreibung
mapP	Wendet Funktion auf alle Element an
filterP	Erstellt Array aller Elemente, die ein bestimmtes Prädikat erfüllen
sumP	Reduziere mit (+)
(+:+)	Konkateniere zwei Arrays
zipWithP	Zwei Arrays gemäß Verbindungsfunktion zusammenführen
anyP	Finde heraus, ob irgendein Element das Prädikat erfüllt
concatP	Konkateniere alle Unterarrays
nullP	Test, ob das Array leer (= [: :]) ist
lengthP	Länge des Arrays
(!:)	array !: i gibt das i-te Element von array zurück (Index fängt bei 0 an)

Bei der Verwendung von solchen parallelen Arrays ist zu beachten, dass sich die Auswertungsstrategie von der bei „normalen“ Listen oder Arrays unterscheidet. Während normalerweise jedes Element unabhängig von den anderen genau dann ausgewertet wird, wenn es benötigt wird, werden sobald das ein Element eines parallelen Arrays benötigt wird *alle* Elemente (parallel) berechnet. Das ist auch wichtig für eine sinnvolle Verwendung, da sonst, je nach Zugriffsmuster, die Auswertung doch vollständig sequentiell sein könnte.

Mit diesen Konstrukten ist es nun sehr einfach möglich Programme mit einem hohen Parallelitätsgrad zu entwickeln.

Beispiel 4. Ein sehr einfaches Beispiel, ist die folgende Matrix-Vektor-Multiplikation aus [2].

```
type Vector = [:Float:]
type Matrix = [:Vector:]

vecMul :: Vector -> Vector -> Float
vecMul a b = sumP [: x*y | x <- a | y <- b :]

matMul :: Matrix -> Vector -> Vector
matMul m v = [: vecMul r v | r <- m :]
```

Hier sind die vorliegenden Daten sehr regulär, jeder Vektor hat die gleiche Länge und der Berechnungsaufwand für jedes Element im Ergebnis ist nahezu identisch. Auch moderne Fortran-Compiler sind in der Lage derartige Berechnungen automatisch zu parallelisieren.

Beispiel 5. Ein etwas komplizierteres Beispiel (ebenfalls aus [2]) ist die folgende Implementierung der Multiplikation von dünn besetzten Matrizen, die als Arrays von Spalten

dargestellt werden, wobei die Zeilen Arrays von Index-Wert-Paaren sind, wobei der Index die Spalte des Werts angibt. Alle Elemente, die in einer Zeile nicht auftreten werden als Null angenommen.

```
1 type SparseVector = [: (Int, Float) :]
2 type SparseMatrix = [: SparseVector :]
3
4 sparseVecMul :: SparseVector -> Vector -> Float
5 sparseVecMul sv v = sumP [: x * v!:i | (i,x) <- sv :]
6
7 sparseMatMul :: SparseMatrix -> Vector -> Vector
8 sparseMatMul sm v = [: sparseVecMul r v | r <- sm :]
```

(!:) ist der Array-Indexoperator, der zu einem gegebenen Index i das i-te Element array !: i des Arrays array liefert.

Hierbei können nun die Längen und damit auch der Berechnungsaufwand der einzelnen Zeilen sich erheblich unterschieden, je nach dem, wie viele Werte 0 sind. Die Regularität, die Fortran oder auch CUDA voraussetzen ist hier also nicht gegeben.

Zur Implementation wird hier von den bei Haskell ansonsten üblichen Formen der Datenrepräsentation abgerückt und stattdessen die Daten eines parallelen Arrays von einigen internen Verwaltungsinformationen, in einem zusammenhängenden Speicherbereich abgelegt, wie man das bei Sprachen wie C oder Fortran, die sich im Bereich des High Performance Computing großer Beliebtheit erfreuen, tun würde. Das macht die Aufteilung der Berechnungen auf verschiedene Kerne besonders leicht, da die gesamten Daten einfach in (so weit wie möglich gleichgroße) Blöcke zerlegt werden, von denen jeder verwendete Kern einen Bearbeitet. Insbesondere zur Ausnutzung moderner Caching-Mechanismen ist dies ein ausgesprochen wichtiger Aspekt von Data Parallel Haskell, ohne den eine gute Performance in aller Regel nicht möglich wäre.

Dafür musste einiges am Unterbau von GHC verändert werden. Will man beispielsweise effizient parallele Arrays von n Paaren verarbeiten, so ist es erheblich besser, statt hintereinander n Paare mit den dazugehörigen Metadaten in den Speicher zu schreiben, die Daten als Paar zweier solcher zusammenhängender Blöcke zu speichern. Dazu ist es aber nötig bei polymorphen Typen für bestimmte Instanzen eine andere Implementierung anzugeben, was ursprünglich bei Haskell nicht vorgesehen war. Dies ist wohl einer der Gründe dafür, dass sich Data Parallel Haskell auch heute, nach mehreren Jahren der Entwicklung, noch nicht fertig ist.

Mit Data Parallel Haskell lassen sich also gewisse inhärent parallele Strukturen ausnutzen ohne zusätzlichen Aufwand nutzen. Dabei muss man sich als Programmierer auch nicht den Kopf darüber zerbrechen, was zum Beispiel passiert, wenn parallele Datenstrukturen andere parallele Datenstrukturen enthalten, was bei fast allen anderen Sprachen mit Unterstützung für Datenparallelismus der Fall wäre.

Für die hier verwendeten Transformationen ist es ebenfalls ungemein wichtig, dass die verwendeten Funktionen keine Nebeneffekte haben können. Sonst könnte es ja bei mapP f list passieren, dass das die Berechnung von f angewandt auf das erste Listenelement eine Änderung des Ergebnisses der anderen Aufrufe von f verursachen würde. Damit

müsste man aber um sicherzustellen, dass dasselbe Ergebnis wie bei der sequentiellen Berechnung herauskommt, die Liste ebenfalls sequentiell abarbeiten.

4 Andere Erweiterungen und ähnliche Sprachen

Es gibt noch viele weitere Haskell-Erweiterungen zur Parallelisierung von Programmen, von denen hier einige erwähnt werden sollen.

Eden lässt den Programmierer explizit Kommunikationsstrukturen auf einem hohen Level angeben. Um eine Funktion parallel auszuführen wird dieser in einem Prozess verpackt, der die angegebene Berechnung im Hintergrund ausführt, sobald alle benötigten Parameter vorliegen. Diese werden vom Master-Thread berechnet. Die Übertragung der Daten erfolgt im einfachsten Fall über implizit definierte Kanäle, der Benutzer kann aber auch explizit angeben, was für Kanäle es gibt, dass heißt welche Prozesse durch sie verbunden werden.

Eden läuft auf verteilten Rechnern und benutzt zur Implementierung der zugrundeliegenden Kommunikation Nachrichtenaustausch (entweder in Form der Parallel Virtual Machine (PVM) oder des Message Passing Interface (MPI)).

Als Laufzeitumgebung kommt eine Erweiterung des Systems von GpH zum Einsatz.[3, 4]

Glasgow Distributed Haskell ist eine Obermenge von Glasgow Parallel Haskell und Concurrent Haskell, wobei Programme verteilt per PVM ausgeführt werden. Im Gegensatz zu Eden erfolgt die Kommunikation aber aus Sicht des Programmierers wie mit den Konstrukten, die in Concurrent Haskell verwendet werden.[5]

Parallel Haskell (pH) ist eine Variante von Haskell, die konsequent Eager Evaluation einsetzt. Das führt dazu, dass einige in Haskell unproblematische Konstrukte wie die Funktion

```
1 takeNats n = take n [1..]
```

die eine Liste der ersten `n` natürlichen zurückliefert, in pH nicht terminieren. In pH sind verschiedene Konzepte aus der Datenfluss-Sprache *Id*, sowie imperative Konzepte wie Schliefen eingeflossen.[6]

Eager Haskell basiert zum Teil auf den gleichen Konzepten, wie pH, verwendet aber statt Eager Evaluation eine spekulative Auswertung, dass heißt Ausdrücke werden so weit wie möglich im Hintergrund ausgewertet, auch wenn die Ergebnisse möglicherweise nicht benötigt werden. Dadurch, dass die Ausdrücke keine Nebeneffekte haben, können nicht benötigte Ausdrücke ohne Probleme ignoriert werden. Die dabei berechneten Daten können, sobald sie sicher nicht mehr benötigt werden, vom Garbage Collector entfernt werden.

Dadurch ist es möglich auch Haskell-Programme, die bei pH zu Nichtterminieren führten in Eager Haskell sinnvoll zu verwenden. Dauert die Berechnung an einer Stelle zu lange, so wird sie unterbrochen und nur dann fortgesetzt, wenn sie benötigt wird. Dadurch entfällt in den meisten Fällen der Overhead, den Lazy Evaluation üblicherweise mit sich bringt, ohne dass man auf die Vorzüge, etwa bei der Arbeit mit unendlichen Datenstrukturen verzichten müsste. Es wird zwar in gewissen Fällen mehr ausgewertet als nötig ist, doch bei ausreichend hohem Parallelitätsgrad ist das Programm mit der gesamten Berechnung dennoch schneller fertig.[7]

5 Schluss

Insgesamt bietet Haskell eine große Auswahl grundverschiedener Parallelisierungsmethoden, von explizitem Parallelismus mit Concurrent Haskell über den annotationsartigen Ansatz von Glasgow parallel Haskell mit `par` und `pseq` bis zur impliziten Parallelisierung mittels Data Parallel Haskell.

Data Parallel Haskell bietet etwas, das es sonst abgesehen vom proof-of-concept in NESL nur in dem ML-basierten Forschungsprojekt *Manticore* gibt, dass ähnliche Ziele verfolgt und auch rein funktional Arbeitet. Gerade bei der Verwendung sehr hoher Parallelitätsgrade, die zweifellos in Zukunft unumgänglich sein wird, kommt man an Datenparallelismus praktisch nicht vorbei. Hier ist insbesondere eine über flachen Datenparallelismus hinausgehende Unterstützung ungemein hilfreich, da so sequentiell formulierte Programme in vielen Fällen ohne große Änderungen am Quelltext in parallele überführt werden können.

Zusätzlich zu den hier vorgestellten Ansätzen gibt es natürlich auch die Möglichkeit mit Hilfe von entsprechenden Bindings mit Haskell Bibliotheken wie MPI (etwa in Kombination mit Concurrent Haskell) zu verwenden, so dass man hier auch viele Optionen, die man bei Verwendung von C oder Fortran hat, benutzen kann. Im Hinblick auf die von Haskell zur Verfügung gestellten Möglichkeiten wird man dies aber wohl in aller Regel nicht tun wollen.

Literatur

[1] B. O'Sullivan, D. Stewart, J. Goerzen: *Real World Haskell*, http://book.realworldhaskell.org/read/concurrent-and-multicore-programming.html (29. 1. 2012).

[2] S. Peyton Jones and S. Singh: *A Tutorial on Parallel and Concurrent Programming in Haskell*, Advanced Functional Progamming Summer School, Nijmegen, Mai 2008, http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/AFP08-notes.pdf (26. 1. 2012).

[3] Eden Homepage, http://www.mathematik.uni-marburg.de/~eden/ (26. 1. 2012).

[4] R. Loogen: *Eden - Parallel Functional Programming in Haskell*, CEFP Summer School, Budapest, Ungarn, Juni 2011, http://www.mathematik.uni-marburg.de/~eden/paper/edenCEFP.pdf (26. 1. 2012).

[5] Glasgow distributed Haskell Homepage, http://www.macs.hw.ac.uk/~dsg/gdh/ (29. 1. 2012).

[6] Parallel Haskell Homepage, http://csg.csail.mit.edu/projects/languages/ph.shtml (29. 1. 2012).

[7] Eager Haskell Homepage, http://csg.csail.mit.edu/pubs/haskell.html (24. 1. 2012).

[8] Haskell-Homepage: http://haskell.org.

[9] GHC-Homepage: http://haskell.org/ghc/.

[10] Real World Haskell: http://book.realworldhaskell.org.

[11] Data Parallel Haskell: http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell.

[12] Vortrag von S. Peyton Jones über Data Parallel Haskell: http://www.youtube.com/watch?v=NWSZ4c9yqW8.