Seminar Programmiersprachen und Programmiersysteme

Nested Data Parallelism in Haskell

Felix Magedanz

2. Juni 2010

Betreuer: Dr. Frank Huch

Inhaltsverzeichnis

1 Einleitung

Bei der Entwicklung von parallelen Programmen für Mehrkern-Prozessoren und verteilte Architekturen bietet sich eine funktionale Programmiersprache wie Haskell an, da die Gefahr von (unerwünschten) Seiteneffekten hier per Definition nicht gegeben ist. Seiteneffektbasiertes Programmieren (mit imperativen Programmiersprachen wie C, C++, Java) wird bei paralleler Ausführung auf mehreren Prozessoren schnell schwer zu handhaben und bedarf spezieller Synchronisations- und Kommunikationselemente, die oft vom Programmierer eigenhändig in die Programme integriert werden müssen.

Ein Ansatz, der aus der Welt der *Massiv-parallelen Computer* – den sogenannten *Super-Computern*, z.B. für meteorologische oder geographische Berechnungen – kommt, ist die Datenparallelität (*data parallelism*). Mit diesem Verfahren werden parallele Berechnungen auf (großen) Datenmengen möglich.

Diese bewährte Technik soll durch die Erweiterung des Haskell-Compilers GHC um spezielle Konstrukte zur Programmierung von datenparallelen Operationen in der Sprache Haskell nutzbar gemacht werden.

Das Team um Simon Peyton Jones – die Entwickler der Sprache Haskell – beschäftigen sich seit einigen Jahren mit der Entwicklung von *Data Parallel Haskell* (DPH) und haben bereits diverse Arbeiten zu diesem Thema veröffentlicht (u.a. [4, 10], auf denen die vorliegende Arbeit größtenteils basiert).

In Kapitel 2 werden kurz die zum Verständnis nötigen Grundlagen geschaffen. In 3 werden die Sprach-Erweiterungen vorgestellt, die DPH möglich machen und anhand einiger Beispiele erläutert. Um die in DPH geschriebenen Programme mit GHC kompilieren zu können sind einige Verfahren nötig, die ab 3.2 und in 4 und 5 behandelt werden.

2 Grundlagen

In diesem Kapitel soll zunächst das grundlegende Konzept der *Datenparallelität (data parallelism)*[7] vorgestellt werden. Als die klassische Umsetzung dieses Konzeptes gilt die Sprache NESL[1], die Anfang der 90er Jahre von Guy E. Blelloch und seinen Mitarbeitern an der Carnegie Mellon University in Pittsburgh, USA entwickelt wurde. NESL vereint die die Effizienz und die Skalierbarkeit des *flat data parallelism* mit dem flexiblen Programmierschema des *nested data parallelism.*[1] Die Umsetzung von *nested data par-*

[1] Siehe [10]: *Abstract*, S. 383

allelism in NESL war Vorlage für die Entwicklung von Data Parallel Haskell[2] und soll ebenfalls in diesem Kapitel näher betrachtet werden. Ausserdem wird kurz die Sprache Haskell vorgestellt.

2.1 Datenparallelität

Eine Form des Parallelismus bei Berechnungen findet auf der Ebene der Daten statt. Die Architekturen, auf denen *data parallelism* ausführbar ist, sind SIMD (*single instructions, multiple data*) und MIMD (*multiple instructions, multiple data*). Es wird parallelisiert, indem die gesamte Datenmenge auf mehrere berechnende Einheiten (mehrere CPUs in einem Multiprozessor-System bzw. mehrere Maschinen in einem verteilten System) verteilt wird. Diese berechnenden Einheiten führen jeweils die gleichen Instruktionen auf unterschiedlichen Teilen der Daten aus. Die Koordination kann durch einen oder mehrere extra-Prozess(e) erfolgen, die Instruktionen können aber auch derart gestaltet sein, dass eine externe Koordination überflüssig wird. Auch die Kommunikation der berechnenden Einheiten untereinander ist möglich.[3]

Der Datenparallelität steht die *Taskparallelität* gegenüber. Hier arbeiten verschiedene Instruktionsfolgen parallel auf entweder den selben Daten (MISD, *multiple instructions, single data*) oder auf verschiedenen Daten (wieder: MIMD). Die Variante MISD hat praktisch wenig Relevanz, während MIMD die zur Zeit am meisten verbreitete Technologie ist (z.B. Desktop-Mehrkern-Prozessoren etc.).[4]

Im Zusammenhang mit Programmiersprachen, die in der Lage sind, datenparallele Algorithmen zu implementieren, bietet sich eine weitere Unterteilung in *flat data parallelism* und *nested data parallelism* an.

Flat Data Parallelism

Mit *flat data parallelism* wird das parallele Berechnen von Daten in einer *flachen* Datenstruktur wie z.B. einem Array bezeichnet. Jede Funktion, die auf die jeweiligen Elemente parallel angewendet werden kann, ist sequentiell und hat die gleiche Berechnungszeit wie die Berechnungen auf den anderen Elementen der (flachen) Datenstruktur.[5]

[2] Siehe [4]: *Abstract*, S. 10
[3] Gesamter Absatz, siehe [7]: S. 1170f
[4] Informationen entnommen aus: [6, 5]
[5] Siehe [10]: *Introduction*, S. 383

Nested Data Parallelism

Diese Form liegt dann vor, wenn jede Funktion (auch eine parallele) auf eine Menge von Daten angewendet werden kann. Ein Beispiel für eine solche Berechnung ist das parallele Aufsummieren der Zeilen einer Matrix, wobei jede Summation als Baum-Summe ebenfalls parallel erfolgt (Abbildung 1).

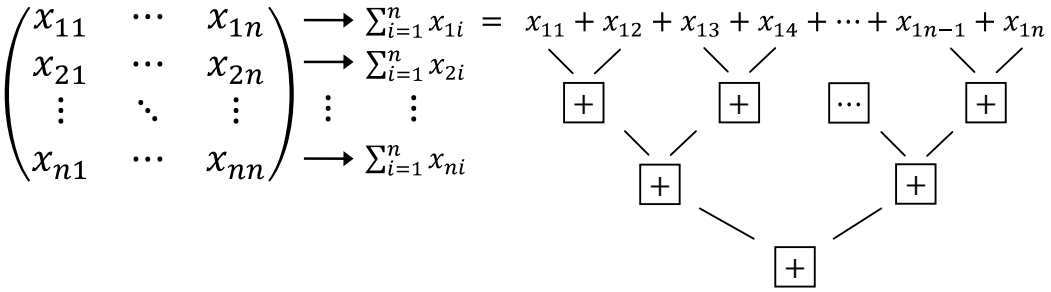

Abbildung 1: Zeilensummen einer Matrix, verschachtelt parallel berechnet

Die Möglichkeit, verschachtelte parallele Operationen benutzen zu können, ist entscheidend, wenn man als Programmierer Algorithmen in der Art implementieren möchte, wie sie intuitiv zu verstehen sind. Jeder *divide-and-conquer*-Algorithmus wäre hier als Beispiel zu nennen.[6]

Es ist möglich, *nested data parallelism* in *flat data parallelism* zu überführen. Dies ist nötig, wenn die verschachtelten Hochsprachen-Konstrukte von einem Compiler in flache – und effizient berechenbare – parallele Anweisungen übersetzt werden. Die Vorgehensweise, Vektorisierung (*vectorisation*) genannt, wird in Kapitel 4 näher behandelt. Erstmals wurde diese Technik in NESL eingeführt, um verschachtelte parallele Programmier-Konstrukte zu ermöglichen.[7]

2.2 NESL

Die Sprache NESL wurde Anfang der 90er Jahre basierend auf ML[8] und SETL[12] entwickelt, unter anderem um implizite Parallelität der Berechnungen für den Programmierer zu ermöglichen. Dieser soll in seinen Programmen spezifizieren, *was* parallel berechnet werden kann, aber nicht, *wo* und *wann* diese Berechnungen stattfinden. Wie

[6] Siehe [1]: *Nested Data Parallelism and NESL*, S. 90
[7] Siehe [1]: *Introduction*, S. 3

bereits angedeutet kann die Parallelität auch verschachtelt angewendet werden. NESL ist eine größtenteils funktionale Sprache und ist damit weitestgehend seiteneffektfrei.[8]

Die Zielplattformen für NESL waren ursprünglich CRAY, Connection Machines und MPI. Die Sprache wird/wurde vorallem für Lehrzwecke verwendet, um die parallelen Konzepte möglichst einfach präsentieren und umsetzen zu können.[9]

Data Parallelism in NESL

NESL unterstützt Datenparallelismus durch Unterstützung von Operationen auf Sequenzen, also eindimensionalen Arrays. Die wichtigste Rolle bezüglich der Datenparallelität übernimmt das *apply-to-each*-Konstrukt, welches durch eine Mengen-basierte Notation implementiert wird. Der Ausdruck

```
{a * a : a in [2, 5, -3, 11]};
```

quadriert jedes Element der der Sequenz `[2, 5, -3, 11]` und liefert `[4, 25, 9, 121]` als Sequenz vom Typ `[Int]` zurück. Informell bedeutet er:

> "in parallel, for each `a` in the sequence `[2, 5, -3, 11]`, square `a`."

Vorauszusetzen ist, dass alle Elemente eines Arrays den selben Typ haben. Das *apply-to-each*-Konstrukt kann auch auf mehrere Sequenzen angewendet werden. So addiert der Ausdruck

```
{a + b : a in [2, 5, -3, 11]; b in [3, 6, -2, 12]};
```

die Elemente der beiden Sequenzen paarweise und liefert `[5, 11, -5, 23]` zurück. Auch eine Filter-Funktion ist möglich. Der Ausdruck

```
{a * a : a in [2, 5, -3, 11] | a > 0};
```

liefert `[4, 25, 121]`. Informell kann er wie folgt beschrieben werden:

> "in parallel, for each `a` in the sequence `[2, 5, -3, 11]` such that `a` is greater than 0, square `a`."

Anzumerken ist, dass die verbleibenden Elemente ihre relative Ordnung behalten.

[8] Siehe [2]: *Nested Data Parallelism and NESL*, S. 90
[9] Siehe [1]: *Introduction*, S. 2

Nested Data Parallelism in NESL

Die besondere Eigenschaft von NESL, verschachtelten Parallelismus zu erlauben, soll anhand der folgenden Beispiele verdeutlicht werden. Eine verschachtelte Sequenz hat die Darstellung `[[2, 1], [7, 3, 0], [4]]` und den Typ `[[Int]]`. Zusammen mit der Unterstützung von Datenparallelismus in Form von Operationen auf parallelen Arrays ergibt sich bei Anwendung der *apply-to-each / in parallel...*-Semantik hier also verschachtelter Parallelismus. Als Beispiel dient hier die parallele Funktion `sum`, angewendet auf die verschachtelte Sequenz:

```
{sum(v) : v in [[2, 1], [7, 3, 0], [4]]};
```

Ausgewertet wird dieser Ausdruck zu `[3, 10, 4]` vom Typ `[Int]`. Parallel ausgewertet wird hierbei nun sowohl jede `sum`, da die Sequenz parallel implementiert ist, als auch die drei Instanzen von `sum` aufgrund der Definition des *apply-to-each*-Konstrukts, welches so definiert ist, dass alle drei Instanzen parallel laufen können.[10]

Als Beispiel für ein (Daten-)paralleles Programm in NESL bietet sich eine Implementierung des *Quicksort*-Algorithmus an (Abbildung 2). Dieser verwendet *divide-and-conquer*. Durch Anwendung des Algorithmus auf eine Sequenz `s` werden die Elemente von `s` auf

```
function quicksort(a) =
   if (#a < 2) then a       -- #a liefert die Länge
   else                     -- der Sequenz a zurück
     let pivot = a[#a/2];
       lesser  = { e in a | e < pivot };
       equal   = { e in a | e == pivot };
       greater = { e in a | e > pivot };
       result  = { quicksort(v) : v in [lesser,greater] };
     in result[0] ++ equal ++ result[1];
```

Abbildung 2: Der Quicksort-Algorithmus in NESL

drei Untermengen (`lesser, equal, greater`) verteilt und es erfolgen rekursive Aufrufe auf den Mengen `lesser$} und {\verb`greater§. Um diese ausführen zu können, werden die beiden Sequenzen in einer verschachtelten Sequenz konkateniert und `quicksort` wird auf die beiden Elemente parallel angewendet. In der letzten Zeile werden die beiden Er-

[10] Aus [1]: *Nested Parallelism*, S. 8

gebnisse der rekursiven Aufrufe extrahiert und mit mit den Elementen aus `equal` in der richtigen Reihenfolge konkateniert (Operator `++`).

Die Berechnung von `lesser, equal, greater` ist datenparallel während in der Berechnung von `result` verschachtelter Datenparallelismus vorliegt.[11]

2.3 Haskell

Haskell ist eine rein funktionale Programmiersprache, die nach dem Mathematiker Haskell B. Curry benannt wurde. Currys Arbeiten in der mathematischen Logik waren die Grundlage bei der Entwicklung der funktionalen Programmiersprachen. Die bekanntesten Werkzeuge für Haskell sind der *Glasgow Haskell Compiler* (GHC) sowie der Haskell-Interpreter *Hugs*.

Auf der Homepage der Entwickler von Haskell[12] ist die folgende Zusammenfassung zu finden:

> "Haskell is an advanced purely functional programming language. An open source product of more than twenty years of cutting edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable high-quality software."

Nach Aussage von Simon Peyton Jones, einem der *Urväter* von Haskell, ist es Haskell gelungen, auch ausserhalb der Forschung eine wichtige Rolle zu spielen. So benutzen mehr und mehr Unternehmen auch für ihre praxisnahe Programmierung die Sprache, was auf die genannten Vorteile der Sprache zurückzuführen sei.[13]

3 Data Parallel Haskell

Die Motivation bei der Entwicklung von *Data Parallel Haskell* (DPH) war es, die Vorteile einer rein funktionalen – aber auch in der Praxis verbreiteten – Programmiersprache wie Haskell zu verbinden mit der Umsetzung des *nested data parallelism* in der Sprache NESL.

[11] Beispiele in Anlehnung an [2]: *Figure 5*, S. 89, sowie *Nested Data Parallelism and NESL*, S.90
[12] http://www.haskell.org
[13] Informationen aus [9]

Als Compiler für DPH dient der Haskell-Compiler GHC. Dieser unterstützt bereits andere Paradigmen des Parallelismus, zum einen expliziten Kontroll-Parallelismus, koordiniert durch *transactional memory*, zum anderen semi-implizite Nebenläufigkeit die auf Annotationen basiert.[14]

Im Folgenden wird die aus NESL übernommene Technik des *nested data parallelism* betrachtet, die DPH ausmacht.

3.1 Programmieren in DPH

Data Parallel Haskell ergänzt Haskell um die folgenden Konstrukte:

- Ein neuer Typ *parallel arrays* mit der Notation `[:e:]` für Arrays vom Typ `e`. Diese Arrays sind indiziert mit Werten vom Typ `Int`. Semantisch entspricht ein Array `[:a:]` weitestgehend einer Liste `[a]`, in der Ausführung sind sie aber verschieden. Ein Array kann Elemente jedes beliebigen Typs enthalten, auch Arrays und Funktionen.
- Eine große Anzahl von *parallel operations*, die auf die parallelen Arrays angewendet werden können. Diese Operationen habend weitestgehend die gleichen Namen wie ihre auf Listen definierten Pendants – sind aber zusätzlich mit dem Suffix `P` versehen.
- Analog zu den *list comprehensions* in Haskell gibt es in DPH *parallel array comprehensions* für die parallelen Arrays.

Eine Eigenschaft der parallelen Arrays ist besonders hervorzuheben: Die Auswertung der Arrays erfolgt strikt. D.h. die Nachfrage nach *einem* Element eines Arrays führt zu der Auswertung *aller* Elemente. In Abbildung 3 sind einige Operationen auf parallelen Arrays aufgeführt[15].

Als Beispiel für das Programmieren mit parallelen Arrays in DPH bietet sich das Skalarprodukt an. Der Typ `[:Float:]` eignet sich für die Repräsentation eines Vektors:

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

Ein Vektor ist also ein paralleles Array mit Elementen vom Typ `Float`. Ein weiterer Typ für (schwach besetzte) Vektoren ist beispielsweise:

```
type SparseVector = [:(Int,Float):]
```

[14] Siehe [4]: *Introduction*, S. 10
[15] Aus [10]: *Introduction*, S. 384

```
(!:)       :: [:a:] -> Int -> a
sliceP     :: [:a:] -> (Int,Int) -> [:a:]
replicateP :: Int -> a -> [:a:]
mapP       :: (a -> b) -> a -> [:a:]
zipP       :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP   :: (a -> b -> c) -> [:a:] -> [:b:] -> [:c:]
filterP    :: (a -> Bool) -> [:a:] -> [:a:]
...
concatP    :: [:[:a:]:] -> [:a:]
concatMapP :: (a -> [:b:]) -> [:a:] -> [:b:]
unconcatP  :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
expandP    :: [:[:a:]:] -> [:b:] -> [:b:]
...
combineP   :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
splitP     :: [:Bool:] -> [:a:] -> ([:a:],[:a:])
```

Abbildung 3: Typ-Annotationen für parallele Operationen auf Arrays

Die Berechnung des Skalarproduktes kann nun wie folgt definiert werden:

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP [: x * (v!:i) | (i,x) <- sv :]


sumP :: Num a => [:a:] -> a
```

Der Ausdruck `[: x * (v!:i) | (i,x) <- sv :]` ist eine *parallel array comprehension* vom Typ `[:Float:]`. Für jedes Element `(i,x)` aus dem (schwach besetzten) Vektor `sv` wird das Produkt von `x` mit dem entsprechenden Element aus dem Vektor `v` berechnet. Durch die Notation `(v!:i)` wird aus `v` das Element an der Stelle `i` referenziert, also ein Wert vom Typ `Float`. Anschließend werden die Elemente des durch die *parallel array comprehension* entstandenen Vektors mit `sumP` (der parallelen Variante von `sum`) aufsummiert.[16]

Die verschachtelung der parallelen Arrays gestaltet sich in DPH ähnlich wie schon in NESL. Beispielsweise die Definition einer (schwach besetzten) Matrix

```
type SparseMatrix = [:SparseVector:]
```

in Verbindung mit der folgenden Definition einer Matrix-Vektor-Multiplikation

[16] Die (parallele) Berechnung von `dotp` folgt der Idee des *gang parallelism*. Mehr dazu in Kapitel 5.2.

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: dotp row v | row <- sm :]
```

liefert uns verschachtelte parallele Berechnungen. Hier wird für jede Zeile `row` der Matrix `sm` das Skalarprodukt `dotp` mit dem Vektor `v` gebildet. Die Operation `dotp` wird parallel auf jede Zeile der Matrix angewendet. Die Verschachtelung entsteht, da `dotp` selber parallel ist.[17]

Wie bereits in Kapitel 2.1 angedeutet, muss diese Form des *nested data parallelism* von dem Compiler transformiert werden, um effizienten parallelen (Zwischen-)Code zu erzeugen.

3.2 Kompilieren von DPH-Programmen

Die Überführung von hochsprachigen, verschachtelten und datenparallelen Programmen ($\rightarrow$ DPH-Programmen) in effizienten *low-level-code* umfasst eine große Anzahl an *source-to-source*-Transformationen. Viele dieser Transformationen sind in GHC enthalten und nicht DPH-speziell.

Die Transformationen, die durch die neue Funktionalität von DPH erforderlich werden, können – Dank der Unterstützung dieser Funktion durch GHC – als *library code* in den Optimierungsprozess von GHC eingebunden werden. Bis auf die Schritte für die Vektorisierung wurde bzw. wird keine Änderung von GHC selber nötig.

Es kann eine grobe Unterteilung in 4 Schritte vorgenommen werden, in denen ein DPH-Programm in ein effizientes parallel berechenbares Programm überführt wird[18]:

1. ***Desugaring*** entfernt den *syntaktischen Zucker*, und reduziert das Programm auf eine *lambda*-Sprache, namentlich GHC's *Core Language*, eine streng getypte Zwischensprache.
2. ***Vectorisation*** überführt alle verschachtelten datenparallelen Operationen in eine flache Struktur.
3. ***Fusion*** optimiert das *Core*-Programm, indem bspw. redundante Synchronisationen zwischen den parallelen Berechnungen entfernt werden, und Arrays, die als Zwischenergebnisse der Berechnungen vorliegen, gelöscht werden.
4. ***Gang Parallelism*** teilt die parallelen Operationen in Teile, von denen jeweils ein Teil für einen *thread* in einer *gang of threads* vorgesehen ist.

[17] Beispiele aus [4]: *What the programmer sees*, S. 11
[18] Aus [10]: *Compiling DPH programs*, S. 390

Der erste Punkt (*Desugaring*) wird in dem folgenden Unterkapitel 3.3 kurz erklärt. Die Vektorisierung wird in Kapitel 4 im Detail betrachtet, und auf die Schritte 3 und 4 wird im Kapitel 5 näher eingegangen.

3.3 Desugaring

Analog zu den Regeln, die für *list comprehensions* gelten, werden die DPH-speziellen *array comprehensions* in normale Funktionen überführt. Dies hat den Vorteil, dass die Vektorisierung, die die verschachtelten parallelen Konstrukte in *flache* und damit berechenbare Strukturen überführt, nicht auf der gesamten Sprache Haskell definiert sein muss, sondern nur auf den Konstrukten der Zwischensprache des (erweiterten) GHC.

Die parallele Berechnung des Kreuzproduktes aus Kapitel 3

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP [: x * (v!:i) | (i,x) <- sv :]
```

würde beispielsweise in die folgende Form überführt werden:

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP (mapP (\(x,i) -> x * (v!:i)) sv)
```

Eine detailliertere Übersicht zu den Regeln, nach denen das *desugaring* durchgeführt wird, ist u.a. in [11] zu finden. Die Regeln für DPH weichen lediglich im Detail ab (zum Beispiel `mapP` statt `map` etc.). Die Regeln in Abbildung 4 geben nur eine grobe Übersicht, ihre Anwendung auf *echte* Programme würde ineffizienten Code erzeugen. in der Implementierung in GHC werden etwas kompliziertere Regeln verwendet.[19]

4 Vektorisierung

Ursprünglich bezeichnet *vektorisieren* den Vorgang, in dem aus einer Funktion, die auf skalaren Werten definiert ist, eine Funktion entsteht, die auf Vektoren von skalaren Werten definert ist. Das Ziel ist hierbei, das parallele Ausführen der Operationen auf Skalaren in Vektoren zu ermöglichen. Gleichzeitig überführt die Vektorisierung in DPH (und auch schon in NESL) *nested* in *flat data parallelism*.

[19] Entnommen aus [10]: *Desugaring array comprehensions*, S. 391

$$
\begin{array}{lrcl}
\textbf{E}\text{xpressions} & e & ::= & \ldots \mid [: e \mid q :] \\
\textbf{Q}\text{ualifiers} & p, q & ::= & x{<}{-}e \mid e \mid p, q \mid p \mid q
\end{array}
$$

$$\mathcal{D}[\![[: e \mid q :]]\!] = \texttt{mapP}\ (\lambda q_v.e)\mathcal{Q}[\![q]\!]$$

$\mathcal{Q}[\![q]\!]$ computes the parallel array of the tuples generated by q

$$
\begin{array}{rcl}
\mathcal{Q}[\![x{<}{-}e]\!] & = & e \\
\mathcal{Q}[\![e]\!] & = & \texttt{if}\ e\ \texttt{then [:():] else [::]} \\
\mathcal{Q}[\![p, q]\!] & = & \texttt{concatMapP}\ (\lambda p_v.\texttt{mapP}\ (\lambda q_v.(p_v, q_v))\mathcal{Q}[\![q]\!])\mathcal{Q}[\![p]\!] \\
\mathcal{Q}[\![p \mid q]\!] & = & \texttt{zipP}\ \mathcal{Q}[\![p]\!]\mathcal{Q}[\![q]\!]
\end{array}
$$

q_v is a tuple of the variables bound by q

$$
\begin{array}{rcl}
(x{<}{-}e)_v & = & x \\
(g)_v & = & () \\
(p, q)_v & = & (p_v, q_v) \\
(p \mid q)_v & = & (p_v, q_v)
\end{array}
$$

Abbildung 4: Regeln für das *desugaring* der *array comprehensions*

Das Prinzip der Vektorisierung soll anhand des folgenden Beispiels[20] verdeutlicht werden. Für die Funktion

```
f :: Float -> Float
f x = x * x + 1
```

wird die *angehobene* (vektorisierte) Funktion `f^` wie folgt erstellt:

```
f^ :: [:Float:] -> [:Float:]
f^ x = (x *^ x) +^ (replicateP n 1)
       where
         n = lengthP x
```

Diese Version benutzt Operationen, die auf Vektoren definiert sind, in diesem Fall

```
(*^) :: [:Float:] -> [:Float] -> [:Float:]
(+^) :: [:Float:] -> [:Float] -> [:Float:]
```

Die Konstante `1` muss ebenfalls vektorisert werden, dies geschieht durch den Aufruf von `replicateP`.

[20] Aus [10]: *Informal overview of vectorisation*, S. 391f

Die Transformation des Rumpfes von `f` lässt sich in 3 Schritte zusammenfassen:

1. Konstanten werden ersetzt durch einen geeigneten Aufruf von `replicateP`
2. Funktionen werden durch ihre angehobenen Versionen ersetzt
3. Parameter können direkt übernommen werden.

Die Definition von `f^` erfüllt offensichtlich die Gleichung `f^ = mapP f`. Die Idee ist nun, dass Aufrufe von `(mapP f)` durch `f^` ersetzt werden. Dadurch ergibt sich aber ein Problem, das anhand der folgenden Definition deutlich wird:

```
g :: [:Float:] -> [:Float:]
g xs = mapP f xs
```

Hier wird nun `(mapP f)` durch `f^` ersetzt

```
g :: [:Float:] -> [:Float:]
g xs = f^ xs
```

Wenn nun aber Aufrufe der Form `(mapP g)` vorkommen, muss `g` ebenfalls vektorisiert werden. Den Schritten 1 - 3 folgend erhält man

```
g^ :: [:[:Float:]:] -> [:[:Float:]:]
g^ xs = f^^ xs
```

Was aber ist nun `f^^` und wie kann es erzeugt werden? Diese Frage hat Blelloch in [1] beantwortet. Es gibt die Möglichkeit, `f^^` durch `f^` zu definieren:

```
f^^ :: [:[:Float:]:] -> [:[:Float:]:]
f^^ xss = unconcatP xss (f^ (concatP xss))
```

Informell:

1. Konkatenation von `xss` zu einem flachen Vektor
2. Anwendung von `f^` auf diesen Vektor
3. Aufteilung des Ergebnisses aus 2. in einzelne Elemente eines Vektors nach Vorgabe des ursprünglichen `xss`

Damit ist es möglich, jede beliebige Verschachtelungstiefe aufzulösen.

Die Implementierungen von `concatP` und `unconcatP` sind – bei passender Repräsentation der Daten – dementsprechend optimiert, dass ihre Laufzeit konstant ist.

4.1 Repräsentation der Arrays im vektorisierten Code

Um eine möglichst performante Repräsentation der *parallel arrays* in DPH zu erhalten ist es unumgänglich, sich von der Implementierung der parametrischen Arrays in Haskell zu lösen.[21] Ein Array in Haskell ist ein Array mit Zeigern, die wiederrum auf Daten (z.B. Float-Werte) zeigen. Diese Flexibilität bringt einige Nachteile mit sich. So wird durch die zusätzlichen Verweise mehr (Arbeits-)Speicher verbraucht und die Zahl der Speicherzugriffe steigt zwangsläufig. Das größte Problem (aus performance-orientierter Sicht) ist aber, dass die *locality of reference* unter dieser indirekten Speicherung leidet, was vorallem in Verteilten Systemen ins Gewicht fällt.

Um diese Probleme zu lösen, werden für die *parallel arrays* in DPH intern (im vektorisierten Code) die sogenannten *associated data types*[3] verwendet. Diese sind eine nicht-parametrische Repräsentation der Daten (*parallel arrays*), also abhängig von dem Typ der Elemente des Arrays. Ein solches nicht-parametrisches Array wird intern mit `PA` bezeichnet. So entspricht beispielsweise der Typ `PA Int` semantisch dem Typ `[:Int:]`, nur das seine interne Repräsentation eine zusammenhängende Folge von Integer-Zahlen im Speicher ist, während `[Int]` ein Array von Zeigern auf Integer-Zahlen wäre.

Die Umsetzung der *associated data types* erfolgt mit Typklassen, wobei der Typ `PA` hier in Verbindung mit einer Klasse `PAElem` deklariert wird:

```
class PAElem a where
  data PA a
  indexPA     :: PA a -> Int -> a
  lengthPA    :: PA a -> Int
  replicatePA :: Int -> a -> PA a
  -- ... weitere Operationen ...
```

Eine Instanz der Klasse `PAElem` wäre die Implementierung für Integer aus dem Beispiel oben:

```
class PAElem Int where
  data PA Int = AInt ByteArray
  indexPA (AInt ba) i = indexIntArray ba i
  lengthPA (AInt ba) = lengthIntArray ba
  replicatePA n i = AInt (replicateIntArray n i)
  -- ... weitere Operationen ...
```

[21] Siehe [10]: *Representing arrays in vectorised code*, S. 392f

Wie beschrieben wird ein Array mit Integer-Werten repräsentiert als ein fortlaufender Bereich im Speicher (= `ByteArray`).

Die Implementierungen für die anderen primitiven Typen folgen dem gleichen Muster. Komplexere Strukturen haben speziellere Repräsentationen. So wird beispielsweise ein Array von Paaren repräsentiert als ein Paar von Arrays:

```
class (PAElem a, PAElem b) => PAElem (a,b) where
  data PA (a,b) = ATup2 Int (PA a) (PA b)
  indexPA (ATup2 _ arr1 arr2) i = (indexPA arr1 i, indexPA arr2 i)
  lengthPA (ATup2 n _ _) = n
  -- ... weitere Operationen ...
```

Hierbei müssen beide Arrays die gleiche Länge haben, die in dem `Int`-Feld des Konstruktors `ATup2` gespeichert wird. Diese Darstellung führt direkt zu den sehr effizienten Implementierungen für die Operationen `zipPA` und `unzipPA` (analog gibt es zu allen Operationen aus Abbildung 3 die `PA` Varianten):

```
zipPA :: PAElem a => PA a -> PA b -> PA (a,b)
zipPA as bs = ATup2 (lengthPA as) as bs

unzipPA :: PA (a,b) -> (PA a, PA b)
unzipPA (ATup2 _ as bs) = (as,bs)
```

Offensichtlich haben diese Operationen konstante Laufzeit, im Gegensatz zu den äquivalenten Operationen auf Listen, die lineare Laufzeit haben.

Verschachtelte Arrays wie zum Beispiel bei

```
type SparseMatrix a = [:[:(Int,a):]:]
```

werden durch flache Arrays repräsentiert. So kann `PA (PA a)` durch (1.) ein flaches Array `PA a`, das die Daten enthält, und (2.) einen *segment descriptor* vom Typ `PA (Int,Int)`, in dem die jeweiligen Anfangspunkte und Längen der Unterarrays gespeichert sind, dargestellt werden.

Es gibt noch einige weitere spezielle Repräsetationsformen bekannter und häufig auftretender Programmkonstrukte. So gibt es auch für individuelle Datentypen (also benutzerdefinierte) eine Repräsentation im vektorisierten Code. Allerdings entspricht diese in den allermeisten Fällen der *normalen* Darstellung, da alle Typen, in denen keine Funktionen oder parallelen Arrays vorkommen, offensichtlich nicht vektorisiert werden müssen.

4.2 Funktionen

Funktionen setzen sich in ihrer vektorisierten Form aus mehreren Teilen zusammen, die bei dem Prozess der Vektorisierung entstehen. So wird für eine Funktion deren *skalare* (also unveränderte) Definition gespeichert, da diese ebenfalls noch benötigt wird. Zusätzlich wird die angehobene Version vorgehalten. Ausserdem wird eine Umgebung zu jeder Funktion gespeichert, in der Festgehalten wird, welche Werte den freien Variablen der Funktion im Programmkontext zugewiesen sind.

Vektorisierte Funktionen werden als Datentyp deklariert:

```
data (a :-> b) = forall e. PAElem e =>
                 Clo { env  :: e
                     , clo  :: e -> a -> b
                     , clo^ :: PA e -> PA a -> PA b }
```

Damit ist (`:->`) ein algebraischer Datentyp in infix-Notation mit dem Konstruktor `Clo`.

So werden Arrays von Funktionen unterstützt: es muss nur *ein* Zeiger auf den auszuführenden Code mitgegeben werden, die unterschiedlichen Variablenbelegungen ergeben sich durch die Umgebung. Arrays mit Elementen vom Typ (`a :-> b`) haben die Repräsentation

```
Instance PAElem (a :-> b) where
  data (a :-> b) = forall e. PAElem e =>
                   AClo { aenv  :: PA e
                        , aclo  :: e -> a -> b
                        , aclo^ :: PA e -> PA a -> PA b }
  lengthPA (AClo env f f^) = lengthPA env
  indexPA (AClo env f f^) n = Clo (indexPA env n) f f^
  replicatePA n (Clo env f f^) = AClo (replicatePA n env) f f^
```

Bei der Vektorisierung wird also jede Funktion τ_1 `->` τ_2 in eine Funktion τ_1' `:->` τ_2' überführt, wobei τ_1', τ_2' die vektorisierten Versionen von τ_1, τ_2 sind. Weiter muss jeder parametrische Array-Konstruktor `[::]` in den nicht-parametrischen der Form `PA` überführt werden. Als Beispiel bietet sich die folgende simple Funktion an:

```
inc :: Float -> Float
inc = \x -> x + 1
```

Deren vollständig vektorisierte Darstellung hat die Form

```
inc_v :: Float :-> Float
inc_v = Clo () inc inc^

inc :: () -> Float -> Float
inc = \(e, x) -> case e of () -> (+)_v $: x $: 1

inc^ :: PA () -> PA Float -> PA Float
inc^ = \(e, x) -> case e of ATup0 n -> (+)_v $:^ x $:^ (replicatePA n 1)
```

Der Operator `$:` hat die Definition

```
($:) :: (a :-> b) -> a -> b
($:) (Clo env f f^) = f env
```

und dient nur dazu, die *skalare* Version aus der vektorisierten Funktion zu extrahieren. Äquivalent dazu gibt es

```
($:^) :: PA (a :-> b) -> PA a -> PA b
($:^) (AClo env f f^) = f^ env
```

Damit sind also auch Funktionen vollständig vektorisierbar. Es bleibt noch anzumerken, dass ein Quellprogramm optimalerweise nicht vollständig vektorisiert wird, sondern nur die parallelen Stellen überführt werden.

5 Ausführbarkeit auf Multiprozessor-Systemen

Die speziellen Anforderungen, die bei der Ausführung von DPH-Programmen auf einem Multiprozessor-System entstehen, werden in den folgenden *Schritten* weiter berücksichtigt.

5.1 Fusion

Durch den Einsatz von Vektor-Operationen und in dem Vektorisierungsprozess entstehen viele *intermediate arrays*, die Synchronisationspunkte zwischen den *threads* bei paralleler Ausführung darstellen. Viele dieser Arrays sind redundant und werden bei der *Fusion* entfernt. Bei der Funktion `dotP` aus Kapitel 3.3 würden beispielsweise 2 tem-

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP (mapP (\(x,i) -> x * (v!:i)) sv)
```

poräre Arrays entstehen.[22] Wird diese Funktion auf paralleler Hardware ausgeführt, so müssen sich die ausführenden Threads gegenseitig benachrichtigen, wenn sie ihren Teil der Berechnungen auf diesen Arrays beendet haben. Wünschenswert wäre es, wenn jeder Thread[23] seinen Teil komplett abarbeiten könnte und am Ende einfach das Ergebnis aus den Teilergebnissen zusammengesetz werden würde (was dann einen *sinnvollen* Synchronisationspunkt darstellt).

Die Überführung einer solchen *pipeline* aus Berechnungen und Synchronisationspunkten in eine einfache Schleife (je Thread) wird *fusion* genannt.

5.2 Gang Parallelism

Die Berechnungen, die für eine Ausführung – z.B. von `dotp` – nötig sind, werden in *chunks* aufgeteilt. Jeder *chunk* soll berechnet werden – ohne dass die (noch nötigen) Synchronisationspunkte an den temporären Arrays die Berechnung verzögern.

Eine Berechnung wird von einer *gang of threads* ausgeführt, wobei jeder *thread* für genau einen Teil der Vektoren (genau einen *chunk*) zuständig ist. In der Regel gibt es in einer *gang of threads* genau so viele Threads wie das ausführende System berechnende Einheiten hat. Dies erfüllt genau die Anforderungen: Jeder Thread kann seine Berechnungen unabhängig durchführen, und die gesamte Berechnung findet parallel statt und endet – zumindest in der Theorie – gleichzeitig. Das *chunking* wird bei DPH mit *distributed types* umgesetzt. Der Typ `Dist Int` steht beispielsweise für eine Menge von *lokalen* Integer-Werten, so dass jedem Thread einer *gang of threads* ein Wert zugeteilt ist. Genau so verhält sich beispielsweise der Typ `Dist [:Float:]`. Hierbei gibt es für jeden Thread einen lokalen Teil des gesamten Arrays. Die Verteilung des Arrays auf die *gang member* erfolgt mit der Operation

```
splitD :: [:a:] -> Dist [:a:]
```

und das Zusammenfügen der Teile nach den Berechnungen mit

```
joinD  :: Dist [:a:] -> [:a:]
```

[22] Mehr Details dazu in [4]
[23] In Realität sind es *gangs of threads*, mehr dazu in Kapitel 5.2

Es werden auch eine Reihe an Operation auf den verteilten Werten unterstützt. Die Operation

```
mapD :: (a -> b) -> Dist a -> Dist b
```

ist hierbei besonders hervorzuheben, da die Threads in der Regel mit dieser ihre Berechnungen auf ihren Teilen der Daten durchführen.

6 Fazit

Die Fülle an Details, die bei der Implementierung der Librarys für DPH als Erweiterung für GHC bereits umgesetzt wurden, deutet darauf hin, dass die Entwicklung schon sehr weit fortgeschritten ist. Dem praktischen Einsatz dieser Technologie dürfte mit der neusten Version von GHC (6.12.1) nichts mehr im Wege stehen.

Die Entwickler von DPH haben in Ihren Veröffentlichungen zu dem Thema auch gezeigt, dass GHC sehr performante Instruktionsfolgen aus DPH-Programmen erzeugt, die bspw. einer *direkten* Implementierung in `C` sehr nahe kommen.[24]

Problematisch bleibt vielleicht die strikte Auswertung der parallelen Arrays, geht dadurch mit der Lazy-Auswertung ein wichtiger Vorteil von Haskell gegenüber anderen Programmiersprachen verloren. Die Entwickler arbeiten momentan noch an einer Lösung für dieses Problem.[25]

Natürlich lässt sich das Prinzip *data parallelism* nicht auf jedes parallele Problem anwenden, aber die Sprache Haskell bietet auch andere Werkzeuge, wie zum Beispiel Transactional Memory oder eben expliziten Parallelismus.

Die Entwicklung von *Data Parallel Haskell* und die Integration in den Haskell-Compiler GHC bringt der Programmiersprache Haskell einen weiteren Schritt auf dem Weg zu einer verbreiteten und kommerziell eingesetzten Programmiersprache.

[24] Siehe u.a. [4]: *Preliminary Results*, S. 16f
[25] Siehe [10]: S. 406

7 Literatur

[1] Guy E. Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon Univ., September 1995.

[2] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[3] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL*, pages 1–13, 2005.

[4] Manuel M. T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. *DAMP*, 2007.

[5] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(3):5–16, February 1990.

[6] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[7] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12), December 1986.

[8] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[9] Simon Peyton Jones. A taste of haskell. Presentation at the O'Reilly Open Source Convention 2007. Portland, Oregon, July 2007.

[10] Simon Peyton Jones, Roman Leshchinsky, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science (Bangalore)*, pages 383–414, 2008.

[11] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions: comprehensions with order by and group by. In *Haskell Workshop 2007*, pages 61–72, Freiburg, Germany, September 2007.

[12] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Itroduction to SETL*. Springer-Verlag, New York, 1986.