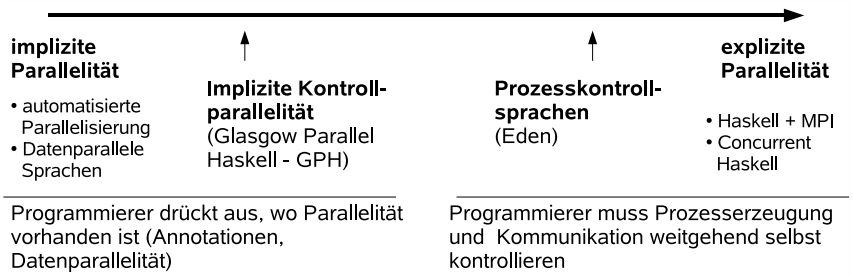
Data Parallel Haskell

Thomas W. Geiger
Betreuung: Prof. Dr. Rita Loogen, Dipl. Inf. Mischa Dieterle
Vortrag im Juli 2009, Marburg

Motivation und Syntax

Data Parallel Haskell (von nun an kurz: DPH) ist eine Spracherweiterung von Haskell um Sprachelemente mit denen Parallelität in Programmen ausgedrückt werden kann. Das Erstellen paralleler Programme gewinnt zunehmend an Bedeutung da momentan die obere Leistungsgrenze für einzelne Prozessoren erreicht zu sein scheint und man daher vermehrt auf Mehrprozessorsysteme, Multicores oder auch verteilte Systeme trifft. Damit gewinnt auch zunehmend die Frage an Bedeutung, wie man in einer Sprache Parallelität zum Ausdruck bringt. Dies kann unterschiedlich explizit geschehen, was die folgende Grafik verdeutlichen soll:



Wie in der Grafik angedeutet, muss bei expliziter Parallelität die Erzeugung von Prozessen, deren Synchronisation, Kommunikation und die sinnvolle Aufgabenverteilung vollständig vom Programmierer übernommen werden - eine schwierige Aufgabe. Wird dagegen als Modell implizite Parallelität verwendet, so parallelisiert der Compiler im Optimalfall das Programm selbstständig: Er erkennt wo Parallelität ausgenutzt werden kann, entscheidet wo dies sinnvoll ist und kümmert sich entsprechend um die Verwaltung der Prozesse. Dazwischen gibt es viele Abstufungen.
Als eine für den Programmierer leicht handhabbare Möglichkeit Parallelität auszudrücken hat sich das Modell der (flachen) Datenparallelität erwiesen. Die Idee ist, gleichzeitig die selbe (sequentielle) Operation auf eine große Menge unterschiedlicher Daten anzuwenden. Die Berechnung kann dann für alle Daten parallel durchgeführt werden, vom System also z.B. bei Multicores auf die vorhandenen Recheneinheiten verteilt werden:

```
-- f sei eine rechenintensive Funktion
f :: Integer -> Integer

x = map f [1..1000]
```

Ein Vorteil der Datenparallelität ist die leichte Verwendbarkeit durch den Programmierer, da immernoch stark sequentiell "gedacht" werden kann. Des weiteren kann Datenparallelität auf vielen Architekturen umgesetzt werden, nicht nur auf Multicores oder Mehrprozessorsystemen sondern auch auf Vektorrechnern, die nur diese Form der Parallelität unterstützen. Schließlich hat sich in der Praxis gezeigt, dass datenparallele Programme gut skalieren wenn man die Anzahl der Recheneinheiten erhöht.
Der Nachteil ist jedoch, dass sich nicht alle Algorithmen mit Datenparallelität parallel formulieren lassen. Einfache Beispiele bieten Algorithmen auf irregulären Strukturen wie dünn besetzten Matrizen oder algebraischen Datenstrukturen in Haskell.
Einen möglichen Ausweg bietet die Idee von Blelloch und Sabot, Datenparallelität flexibler zu machen, indem es zugelassen wird, dass sie verschachtelt auftritt. Wir betrachten folgendes Beispiel, wobei wir (in Vorwegnahme der DPH Syntax) ein Array, welches parallel ausgewertet werden soll, mit `[: :]` notieren:

```
-- f sei eine rechenintensive Funktion
f :: Integer -> Integer

g :: [:[:Integer:]:] -> [:[:Integer:]:]
g = mapP (mapP f)

x :: [:[:Integer:]:]
x = [:[:1, 2, 3:], [:812, 44, 9:], [:24:]:]

-- Aufruf mit verschachtelter Parallelität:
g x
```

Der Aufruf `g x` führt hier zu folgender Reduktion:

```
g x ⇒ mapP (mapP f) [:[:1, 2, 3:], [:812, 44, 9:], [:24:]:]
    ⇒ [: (mapP f [:1, 2, 3:]), (mapP f [:812, 44, 9:]), (mapP f [:24:]) :]
    ⇒ [: [:(f 1), (f 2), (f 3):], [:(f 812), (f 44), (f 9):], [:(f 24):] :]
```

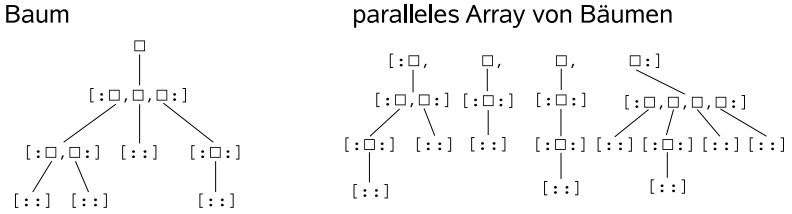
Die parallele Auswertung von g x führt also dazu, dass 3 Recheneinheiten mit der Auswertung folgender Ausdrücke beauftragt werden:

Recheneinheit 1: `[:(f 1), (f 2), (f 3):]`
Recheneinheit 2: `[:(f 812), (f 44), (f 9):]`
Recheneinheit 3: `[:(f 24):]`

Wir bemerken, dass hier zwei Forderungen der Datenparallelität wie sie oben beschrieben wurde (von nun an als flache Datenparallelität bezeichnet) verletzt sind: Zunächst sind die auszuführenden Berechnungen nicht nur in ihren Daten unterschiedlich - flache Datenparallelität fordert zumindest, dass die Berechnungen vergleichbare Zeit benötigen, was hier sicherlich nicht gegeben ist. Außerdem wird gefordert, dass die parallel ausgeführten Berechnungen selbst sequentiell sind. In obigem Beispiel ist aber deutlich zu erkennen, dass die Auswertung auf jeder Recheneinheit selbst parallel erfolgen soll. Damit ist `g x` nicht direkt mit flacher Datenparallelität auswertbar. Bei der verschachtelten Datenparallelität nach Blelloch sollen nun derartige Verschachtelungen gerade zugelassen sein.
Außerdem soll in DPH die Möglichkeit bereitgestellt werden, auch auf algebraischen Datenstrukturen, die sehr irregulär sein können, parallel zu rechnen. Als Beispiel bietet sich eine einfache Baumstruktur an:

```
data Tree a = Node a [:Tree a:]

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Node val [::] = Node (f val) [::]
mapTree f Node val ts   = Node (f val) (mapP (mapTree f) ts)
```

Wünschenswert wäre es, bei einem Aufruf von `mapTree` für einen konkreten Baum oder geliftet für ein Array von Bäumen eine parallele Auswertung der einzelnen Ausdrücke in den Baumknoten zu erreichen.



In DPH ist es für eine solche Struktur jedoch nur möglich "ebenenweise" parallel zu rechnen, im Fall eines Arrays also auf allen Wurzeln parallel, dann auf allen Elementen der ersten Ebene aller Bäume, usf.
Das Ziel von DPH bestehen also darin, verschachtelte Parallelität in Haskell als eine weit verbreitete Sprache

verfügbar zu machen. Diese Erweiterung soll transparent gegenüber den übrigen Möglichkeiten die Haskell bietet, beispielsweise Funktionen höherer Ordnung, sein. Weiterhin wird das Ziel gesetzt, dass DPH von der Architektur nur die Fähigkeit fordert, flach datenparallel rechnen zu können, es lässt also die Möglichkeit zu, auch auf Vektorrechnern implementiert zu werden - obgleich die Implementierung sich momentan auf Multicore-Architekturen konzentriert.
Um diese letzte Forderung umzusetzen ist ein Algorithmus notwendig, welcher ein Programm mit verschachtelter Datenparallelität in eines mit flacher umwandeln kann - der Vektorisierungsalgorithmus. Wir werden später die grundlegenden Ideen dieses Algorithmus vorstellen.
Zunächst soll jedoch die Syntax von DPH angegeben werden. Im wesentlichen wird Haskell dabei um zwei Elemente erweitert:

1. Einen polymorphen Typ: Parallele Arrays vom Typ `e`, notiert durch
   `[: e :]`
2. Parallele Funktionen: Operieren auf parallelen Arrays, z.B.
   `mapP :: (a->b) -> [:a:] -> [:b:]`

Parallele Arrays können dabei weitgehend analog zu den bekannten Listen in Haskell verwendet werden, beispielsweise ist folgender Code möglich:

```
[:1, 2, 3, 4, 5:] :: [:Integer:]
[:sin, cos:] :: Floating a => [:a -> a:]

-- parallel array comprehensions
add1 :: [:Integer:] -> [:Integer:]
add1 xs = [:x + 1 | x <- xs:]
```

Die wesentlichen Unterschiede sind:

- Striktheit: Die Notwendigkeit ein Element eines parallelen Arrays auszuwerten führt zur Auswertung aller Elemente des Arrays
- Ausführungssemantik: Die Auswertung der Elemente eines parallelen Arrays erfolgt parallel

Da parallele Arrays einen neuen Datentyp darstellen, ist es erforderlich, dass DPH zusätzlich noch Funktionen bereitstellt, mit denen solche Arrays manipuliert werden können. Wegen der großen Ähnlichkeit zu Listen, gibt es auch bei diesen Funktionen große Ähnlichkeiten zu den Funktionen von Haskell, die auf Listen operieren. Wir geben eine Auswahl an (ohne Erklärung, die Semantik sollte bei den meisten unmittelbar erkennbar sein):

```
mapP       :: (a->b) -> [:a:] -> [:b:]
zipP       :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP   :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP    :: (a->Bool) -> [:a:] -> [:a:]
lengthP    :: [:a:] -> Int

replicateP :: Int -> a -> [:a:]
concatP    :: [:[:a:]:] -> [:a:]
unconcatP  :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
combineP   :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
expandP    :: [:[:a:]:] -> [:b:] -> [:b:]
```

Auf einige dieser Funktionen gehen wir später detaillierter ein und geben auch Implementierungen an. Als besonders wichtig für die Vektorisierung werden sich die Funktionen `concatP` und `unconcatP` herausstellen, daher betrachten wir folgende Beispiele:

```
x :: [:[:Integer:]:]
x = [:[:1, 2, 3:], [:4, 5:], [:6, 7:]:]

concatP x
   liefert [:1, 2, 3, 4, 5, 6, 7:]

unconcatP [:[:sin, sin, sin:], [:sin, sin:], [:sin, sin:]:] [:1, 2, 3, 4, 5, 6, 7:]
   oder
unconcatP x [:1, 2, 3, 4, 5, 6, 7:]
   liefert [:[:1, 2, 3:], [:4, 5:], [:6, 7:]:]

transposeP x
   liefert [:[:1, 4, 6:], [:2, 5, 7:], [:3:]:]
```

Es handelt sich also, wie die Funktionsnamen andeuten, um Funktionen, die verschachtelte Arrays in flache umwandeln und flache Arrays mit einer Verschachtelungsstruktur versehen. Für letzteres ist natürlich nur irgendein Array notwendig, welches die gewünschte Verschachtelungsstruktur aufweist. Die Typen müssen nicht übereinstimmen, was obiges Beispiel mit dem verschachtelten Array von Funktionen belegt.

Wir wollen uns nun anhand eines komplizierteren Beispiels aus der Praxis davon überzeugen, dass diese wenigen zusätzlichen Sprachelemente ausreichen um selbst komplizierte Algorithmen vollständig parallel formulieren zu können.

Komplexes Beispiel: Das n-Körper-Problem

Als Beispiel zur Verwendung von Data Parallel Haskell formulieren wir den Barnes-Hut Algorithmus zur Approximation des n-Körper-Problems. Dieser ist praktisch relevant und mit einer (flach) datenparallelen Sprache nicht bequem oder nur teilweise parallel formulierbar.
Gegeben seien n Körper im $\mathbb{R}^m$ mit Massen $m_i \in \mathbb{R}_+$. Näherungsweise können wir die Körper als punktförmig annehmen, d.h. sie sind durch

$$q_i \in \mathbb{R}^m \qquad i \in \{1, \ldots, n\}$$

gegeben. Weiterhin sei für jeden Körper seine Geschwindigkeit

$$p_i \in \mathbb{R}^m \qquad i \in \{1, \ldots, n\}$$

zum Anfangszeitpunkt $t = 0$ vorgegeben. Diese zwei Größen genügen in der Formulierung der klassischen Mechanik nach Newton um das System zu bestimmen. Gesucht sind die Bewegungsbahnen

$$\begin{array}{lllll} \text{Position :} & Q_i : & [0,T] \longrightarrow \mathbb{R}^m & & \\ & & t \longmapsto Q_i(t) & \text{mit } Q_i(0) = q_i & \\ & & & & \forall\, i \in \{1, \ldots, n\} \\ \text{Geschwindigkeit :} & P_i : & [0,T] \longrightarrow \mathbb{R}^m & & \\ & & t \longmapsto P_i(t) & \text{mit } P_i(0) = p_i & \end{array}$$

unter der Annahme, dass die Gravitationskräfte aus der klassischen Mechanik wirken. Ein Körper $q_1 \in \mathbb{R}^m$ mit Masse $m_1 \in \mathbb{R}_+$ übt auf einen anderen Körper $q_2 \in \mathbb{R}^m$ mit Masse $m_2 \in \mathbb{R}_+$ die Kraft F_{12} aus, wobei gilt:

$$F_{12} = -G \frac{m_1 m_2}{\|q_2 - q_1\|^2} \frac{q_2 - q_1}{\|q_2 - q_1\|} \in \mathbb{R}^m$$

Dabei ist

$$G = (6,67428 \pm 0,00067) \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$$

die Gravitationskonstante. Die exakte Lösung ist damit durch ein System von Differentialgleichungen bestimmt. Zur näherungsweisen Lösung diskretisieren wir das Zeitintervall $[0, T]$ mit einer äquidistanten Unterteilung mit Schrittweite s und approximieren die Lösung zu den diskreten Zeitpunkten $t_k := s \cdot k \;\; k \in \{1, \ldots, l\}$ (für geeignetes l).

Als naive Idee zur Approximation berechnet man in jedem Zeitschritt für jeden Körper j die Kraft F_{ij}, die jeder andere Körper $i \neq j$ ausübt und summiert diese:

$$F^j := \sum_{\substack{i=1 \\ i \neq j}}^{n} F_{ij}$$

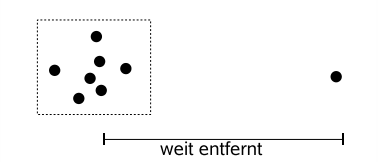
Damit können dann die neuen Positionen und Geschwindigkeiten direkt berechnet werden:

$$\begin{aligned} Q_i(t_{k+1}) &= Q_i(t_k) + P_i(t_k) \cdot s \\ P_i(t_{k+1}) &= P_i(t_k) + \tfrac{F^i}{m_i} \cdot s \end{aligned}$$

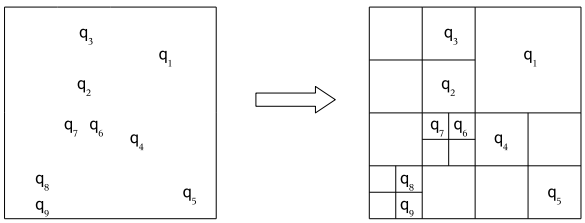
wobei die Newtonsche Formel

$$F = m \cdot a$$

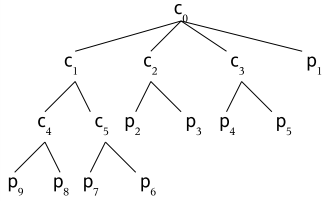
verwendet wurde.
Da in jedem Zeitschritt n^2 Wechselwirkung zu berechnen sind ist die Komplexität $O(n^2)$. Zur Reduktion der Komplexität verwendet man den Barnes-Hut Algorithmus. Die Idee besteht darin, nahe beieinanderliegende Körper aus Sicht eines weit entfernten Körpers gemeinsam zu betrachten, so dass für eine größere Zahl von Körpern nur eine Wechselwirkung zu berechnen ist:



Man kann zeigen, dass hiermit die Komplexität $O(n \cdot log(n))$ erreichbar ist. Der Übersichtlichkeit halber schränken wir uns von nun an auf den Fall $m = 2$, also einen zweidimensionalen Raum ein.
Der Algorithmus besteht damit im wesentlichen aus zwei Teilschritten: Zunächst müssen Gruppierungsinformationen über die Körper berechnet werden. Dazu wird das Gesamtgebiet in dem sich die Körper befinden in vier Teilgebiete aufgeteilt und die Körper anhand dieser Teilbereiche gruppiert. Dieses Vorgehen wird wiederholt bis in einem Bereich höchstens ein Körper enthalten ist.



Dabei wird ein Baum mit den Gruppierungsinformationen erzeugt - die Baumknoten enthalten entweder einen Körper oder die Summe der Massen im Gebiet und das arithmetische Mittel der Orte der Unterbäume.



Der Baum stellt damit die Approximation des Gravitationsfeldes zum berechneten Zeitpunkt dar. In einem zweiten Schritt wird die vom Baum auf alle Körper wirkende Kraft berechnet, indem für einen festen Körper i der Baum von der Wurzel an durchlaufen wird. Ist der Körper vom aktuellen Baumknoten weit genug entfernt,

wird die wirkende Kraft mit den Informationen aus diesem Knoten berechnet. Ansonsten werden rekursiv die von den Unterbäumen wirkenden Kräfte berechnet und summiert.
Zur Implementierung in DPH müssen zunächst die notwendigen Datentypen bereitgestellt werden:

```
type Mass = Float
type Vector = (Float, Float)
type Area = (Vector, Vector)
type Force = Vector
type Velocity = Vector
type Location = Vector

-- beschreibt einen Körper der Simulation
data Particle = Particle { mass :: Mass
                         , location :: Location
                         , velocity :: Velocity
                         }

-- beschreibt den Baum, der die Gruppierungsinformation enthält
data Tree = Node Mass Location [:Tree:]
```

Außerdem stellen wir als Basisoperationen eine Vektoraddition und Skalarmultiplikation zur Verfügung:

```
add :: Vector -> Vector -> Vector
add (x1,x2) (y1,y2) = (x1 + y1, x2 + y2)

mul :: Float -> Vector -> Vector
mul s (x1,x2) = (s*x1, s*x2)
```

Die Implementierung wird drei Hilfsfunktionen benutzen, die den drei Phasen des Algorithmus entsprechen: Eine zum Berechnen der Gruppierungsinformation (`buildTree`), eine zum Berechnen der wirkenden Kräfte (`calcForces`) und schließlich eine Funktion, welche die neuen Positionen und Geschwindigkeiten berechnet (`moveParticles`). Darauf aufbauend wird der Algorithmus in einer Funktion (oneStep) implementiert, welche ausgehend von den Körperdaten zum Zeitpunkt t die Daten zum Zeitpunkt $t + s$ berechnet.
Zunächst stellen wir zur Implementierung von `buildTree` die folgenden Hilfsfunktionen bereit auf deren Implementierung wir verzichten, da sie offensichtlich ist:

```
-- Gibt zurück, ob ein Körper in einem Gebiet liegt oder nicht
inArea :: Area -> Particle -> Bool

-- Zerlegt ein Gebiet in die 4 Teilgebiete
splitArea :: Area -> [:Area:]

-- Berechnet das Zentrum einer Menge von Bäumen
calcCentroid :: [:Tree:] -> (Mass, Location)
```

Die Funktion buildTree selbst hat folgende Definition:

```
-- 1. Argument: Gebiet in dem sich die Körper befinden
-- 2. Argument: Die Körper mit ihren Informationen
-- Resultat: Der Baum mit der Gruppierungsinformation
buildTree :: Area -> [:Particle:] -> Tree
```

Die Implementierung sieht nun wie folgt aus:

```
buildTree :: Area -> [:Particle:] -> Tree
buildTree area [: p :]   = Node (mass p) (location p) [::]
buildTree area particles = Node m l subtrees
                           where
                            (m,l)    = calcCentroid subtrees
                            subtrees = [: buildTree a ps
                                        | a <- splitArea area
                                        , let ps = [:p | p <- particles, inArea a p:]
                                        , lengthP ps > 0 :]
```

Sie wird mit Pattern-Matching rekursiv über die Erzeugung der Unterbäume erklärt. Interessant ist die Definition von `subtrees`. Hier erkennt man deutlich die verschachtelte parallele Auswertung: Die Unterbäume selbst sollen parallel berechnet werden, jedoch auch die Auswertung des Arrays von `splitArea` und des Arrays mit den Körpern in jedem Gebiet sollen parallel erfolgen. Das Aufbauen des Baumes ist der Teil des Barnes-Hut Algorithmus der in flach datenparallelen Implementierungen üblicherweise sequentiell durchgeführt wird, da er nur schwer mit flacher Datenparallelität formuliert werden kann (obgleich es möglich ist).
Als nächstes wollen wir die Hilfsfunktion calcForces zur Berechnung der auf die Körper wirkenden Kräfte implementieren. Diese wird folgende Definition besitzen:

```
-- 1. Argument: Länge des Bereichs in dem sich die Körper befinden; Wird zur Definition, was
--              "weit entfernt" bedeutet benötigt
-- 2. Argument: Der Baum mit der Gruppierungsinformation; Approximiert das Gravitationsfeld
--              welches wirkt
-- 3. Argument: Das parallele Array mit den Körpern, die sich im Gravitationsfeld befinden
-- Resultat: Das parallele Array mit den Kräften, welche auf die einzelnen Körper wirken
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
```

Interessant ist, dass wir die Kräfte ebenfalls in einem parallelen Array zusammenstellen. Dies deutet bereits an, dass in weiteren Rechnungen mit den Kräften parallel weitergearbeitet werden soll. Zur Implementierung stellen wir zunächst wieder einige Hilfsfunktionen zusammen:

```
-- Berechnet die Kraft, die auf einen Körper (1. Argument) von der Masse am gegebenen Ort
-- (2. und 3. Argument) ausgeübt wird nach dem Newtonschen Gravitationsgesetz
forceOn :: Particle -> Mass -> Location -> Force

-- Summiert Kräfte
sumForces :: [:Force:] -> Force
sumForces fs = foldP add (0,0) fs

-- Entscheidung was "weit entfernt" bedeutet; Lassen wir offen. Dazu ist (1. Argument)
-- die Länge des Gesamtbereichs in dem sich die Körper befinden erforderlich
isFar :: Float -> Location -> Particle -> Bool
```

Wieder ist die Implementierung offensichtlich. In der Funktion `sumForces` sieht man, wie diese Aufgabe leicht mit von DPH bereitgestellten Primitiven gelöst werden kann. Was "weit entfernt" bedeutet ist eine Frage, die bei der Konstruktion des Algorithmus beantwortet werden muss und welche selbstverständlich sequentiell implementiert wird - daher ist sie für unsere Betrachtungen nicht relevant und wird offengelassen. Mit dieser Vorarbeit lässt sich nun eine erste Version der Hilfsfunktion `calcForces` direkt formulieren:

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len tree ps = mapP (calcForce len tree) ps

calcForce :: Float -> Tree -> Particle -> Force
calcForce len (Node m l ts) p
        | (isFar len l p) || (lengthP ts = 0) = forceOn p m l
        | otherwise                           = sumForces [: calcForce len t p | t <- ts :]
```

Zunächst wird mit der Funktion `calcForce` erklärt, wie man für einen einzelnen Körper bei gegebenem Gravitationsfeld (Baum) die wirkende Kraft berechnet. Hier findet sich die Approximation des Barnes-Hut Algorithmus wieder: Ist der Körper von der Wurzel des Baumes weit genug entfernt, werden die gemittelten Informationen aus dem Baumknoten zur Kraftberechnung verwendet, ansonsten werden die Kräfte die von den Unterbäumen wirken rekursiv berechnet und summiert. Um damit die Kräfte zu berechnen, die auf ein paralleles Array von Körpern wirken muss nur noch diese Funktion `calcForce` parallel auf alle Arrayelemente angewendet werden. Da in `calcForce` ein paralleles Array auftritt liegt insgesamt eine verschachtelte datenparallele Berechnung vor.

Dieses Beispiel bietet sich an, um auf ein Problem beim Erstellen paralleler Programme aufmerksam zu machen: Obwohl DPH den Programmierer von der Aufgabe befreit sich mit dem Erzeugen und Synchronisieren verschiedener Prozesse auseinanderzusetzen und deren Kommunikation aufzubauen, kann dennoch nicht rein sequentiell gedacht werden. In unserer Implementierung haben wir die Methode `calcForce` zu einer Methode für parallele Arrays geliftet und ihr zuvor den Baum mitgegeben (ein Closure). Der Baum ist damit potentiell für die Berechnung der wirkenden Kraft für jeden der Körper im Array notwendig. Wenn diese Berechnung parallel ausgeführt wird, muss also der Baum potentiell vollständig auf allen Recheneinheiten zur Verfügung stehen was unter Umständen einen hohen Bedarf an Kommunikation erfordert. Eine andere Möglichkeit, `calcForces` zu implementieren ist die folgende:

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len (Node m l ts) ps
          = let
             far_forces    = [: forceOn p m l | p <- ps, isFar len l p :]
             near_ps       = [: p | p <- ps, not (isFar len l p) :]
             near_forces_s = [: calcForces (len / 2) t near_ps | t <- ts :]
             near_forces   = [: sumForces p_forces | p_forces <- transposeP near_forces_s :]
            in
             combineP [:isFar len l p | p <- ps:] far_forces near_forces
```

Hier erfolgt die parallele Berechnung entlang der Baumstruktur und die Teilchen werden entsprechend ihrer Entfernung jeweils aufgeteilt. Insgesamt kann man einsehen, dass der Bedarf an Kommunikation in dieser Implementierung viel geringer ist.
Trotz der hohen Abstraktion die DPH zur Formulierung paralleler Programme zur Verfügung stellt muss der Programmierer also bedenken, in welchem Umfang Daten kommuniziert werden müssen um zu garantieren, dass die Implementierung eines Algorithmus effizient ist.
Wir haben nun noch die Implementierung von `moveParticles` anzugeben. Diese besitzt folgende Deklaration:

```
-- 1. Argument: Schrittweite s
-- 2. Argument: Paralleles Array mit den Eigenschaften der Körper zum Zeitpunkt t
-- 3. Argument: Paralleles Array mit den auf die Körper zum Zeitpunkt t wirkenden Kräften
-- Resultat: Paralleles Array mit den Eigenschaften der Körper zum Zeitpunkt t+s
moveParticles :: Float -> [:Particle:] -> [:Force:] -> [:Particle:]
```

Die Implementierung erfolgt analog zur ersten Version von `calcForces` folgendermaßen:

```
moveParticles :: Float -> [:Particle:] -> [:Force:] -> [:Particle:]
moveParticles ts ps fs = zipWithP (moveParticle ts) ps fs

moveParticle :: Float -> Particle -> Force -> Particle
moveParticle timestep
             Particle { mass = m
                      , location = loc
                      , velocity = vel}
             force
             = Particle { mass = m
                        , location = newloc
                        , velocity = newvel}
              where
               accel = (1/m) `mul` force
               newloc = loc `add` (timestep `mul` vel)
               newvel = vel `add` (timestep `mul` accel)
```

Auch hier geben wir zunächst an, wie man die neuen Daten für einen einzelnen Körper berechnet und liften diese Funktion anschließen zu einer Funktion für parallele Arrays. Offenbar ist diese Implementierung insgesamt flach Datenparallel. Jedoch müssen nicht mehr Daten kommuniziert werden als unbedingt nötig, da zusätzlich zu den zwingend erforderlichen Daten über den Körper und die wirkende Kraft nur die Schrittweite allen Recheneinheiten bekannt sein muss.
Wir haben nun alle Hilfsfunktionen bereitgestellt um die Implementierung des gesamten Algorithmus angeben zu können:

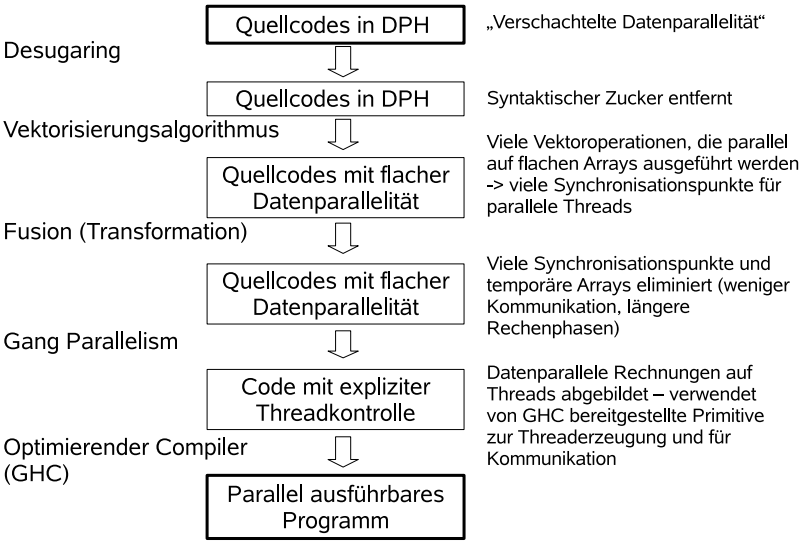
```
-- 1. Argument: Gebiet in dem sich die Körper befinden
-- 2. Argument: Schrittweite s
-- 3. Argument: Paralleles Array mit den Eigenschaften der Körper zum Zeitpunkt t
-- Resultat: Paralleles Array mit den Eigenschaften der Körper zum Zeitpunkt t+s
oneStep :: Area -> Float -> [:Particle:] -> [:Particle:]
oneStep area time particles = moveParticles time particles forces
                              where
                               tree = buildTree area particles
                               forces = calcForces (lengthOf area) tree particles
```

Wobei hier `lengthOf :: Area -> Float` eine Hilfsfunktion ist, die die Länge eines Gebietes berechnet.

Zusammenfassung: Wir haben den Barnes-Hut Algorithmus aufgeteilt in drei Berechnungsphasen implementiert und konnten in jeder der drei Phasen parallel rechnen. In der ersten Phase, dem Aufbauen des Baumes, war es für uns entscheidend, dass wir verschachtelte Datenparallelität verfügbar hatten. Die zweite Phase, das Berechnen der wirkenden Kräfte, haben wir ebenfalls verschachtelt datenparallel implementiert, und dabei festgestellt, dass die konkrete Implementierung gegebenenfalls einen entscheidenden Einfluss auf die notwendige Kommunikation und damit die Effizienz nimmt. Die letzte Phase, das Berechnen der neuen Körperdaten, konnte schließlich völlig befriedigend rein unter Verwendung flacher Datenparallelität implementiert werden. Außerdem haben wir gesehen, dass die einfache Syntax von DPH uns die Möglichkeit bot, den Algorithmus insgesamt mit einem geringen Aufwand sehr elegant zu formulieren.

Implementierung von DPH

DPH wird als eine Reihe von Programmtransformationen im GHC implementiert. Es nutzt momentan vom GHC intern bereitgestellte Primitive zur Threaderzeugung, Synchronisation und Kommunikation, welche auf vielen Plattformen verfügbar sind. Da hiermit ein Programm mit expliziter Threadkontrolle erzeugt wird, können DPH Programme momentan nicht auf echten Vektorrechnern ausgeführt werden. Die Abbildung des flach datenparallelen Programms auf die Primitive des GHC ist jedoch der letzte Transformationsschritt und damit prinzipiell gegen Abbildungen auf andere Architekturen austauschbar. Folgende Transformationen durchläuft ein DPH-Programm:



Das Desugaring löst syntaktischen Zucker auf, beispielsweise die parallelen array comprehensions und ist als Transformation nicht besonders interessant, da es weitgehend mit der entsprechenden Transformation von GHC für Standard-Haskell Programme übereinstimmt. Der zweite Schritt ist für uns besonders interessant: Der Vektorisierungsalgorithmus erzeugt aus einem Programm mit verschachtelter Datenparallelität eines mit flacher. Diese Transformation, in der sich die Kernideen von DPH wiederfinden, wird wir im folgenden näher untersucht. Wir müssen uns jedoch darauf beschränken die grundlegenden Ideen vorzustellen, da der Algorithmus insgesamt sehr kompliziert ist und viele subtile Probleme zu lösen hat, unter anderem solche die durch die Implementierung im GHC entstehen.
Nach der Vektorisierung liegt ein flach datenparalleles Programm vor, welches viele elementare Vektoroperationen verwendet und daher nur kurze parallele Rechenzeiten aufweist, dafür aber einen hohen Bedarf an Kommunikation und Synchronisation. Für die Effizienz eines DPH Programms ist es daher ganz wesentlich, dass sich an die Vektorisierung eine weitere Transformation anschließt - mit Fusion bezeichnet - welche Synchronisationspunkte und temporäre Arrays eliminiert und dadurch parallele Rechenphasen verlängert und notwendige Kommunikation verringert. Die Techniken die hier zum Einsatz kommen sind nicht spezifisch für DPH. Sie werden unabhängig von diesem Projekt schon lange erforscht, da sie sich auf beliebige flach datenparallele Programme anwenden lassen. Man hat festgestellt, dass solche Techniken in einer funktionalen Sprache einfacher zu realisieren sind als in einer imperativen. Es ist jedoch leider nicht möglich im Rahmen dieser Arbeit auf Details einzugehen.
Im darauffolgenden Schritt wird das flach datenparallele Programm auf die vom GHC bereitgestellten Primitive abgebildet. An dieser Stelle erfolgt die Abbildung des Programms auf die Architektur der Zielmaschine. Momentan sind dies im wesentlichen Multicore- und Multiprozessor-Maschinen. Durch Austauschen dieser Transformation gegen eine andere geeignete wären DPH Programme aber prinzipiell auch auf Vektorrechnern ausführbar.
Nach dieser Abbildung liegt ein reguläres Haskell-Programm vor, welches auf GHC-Primitive zurückgreift um Threads zu erzeugen und zu synchronisieren. Ein solches Programm kann der GHC ohne weitere Veränderung direkt optimieren und übersetzen.

Üblicherweise muss von einem Programm nur ein gewisser Teil vektorisiert werden. Beispielsweise lohnt es sich nicht die Teile, die eine grafische Oberfläche verwalten zu parallelisieren. Bestimmte Teile eines Programms lassen sich auch überhaupt nicht vektorisieren, beispielsweise solche die Ein- und Ausgabe verwalten. Daher wurden Techniken zum sogenannten partiellen Vektorisieren entwickelt. Wir können hierauf nicht näher eingehen wollen aber dennoch einige Bemerkungen festhalten: Vektorisierung stellt eine massive Transformation der Quellcodes dar. Wir werden später sehen, dass es notwendig ist, sowohl Funktionen als auch Typen und algebraische Datenstrukturen zu vektorisieren.

$$\begin{array}{lrcl} \mathcal{V}_t[\![\,]\!]: & \text{Type} & \longrightarrow & \text{Type} \\ \mathcal{V}[\![\,]\!]: & \text{Function} & \longrightarrow & \text{Vect.Function} \end{array}$$

Im Falle von Funktionen wird das bedeuten, dass ein neuer algebraischer Datentyp eingeführt wird (`:->`) wel-

cher vektorisierte Funktionen beschreibt. Auch die Repräsentation paralleler Arrays muss sorgfältig gewählt werden um die effiziente Implementierung von DPH zu ermöglichen und wird eine massive Umstrukturierung der Quellcodes bedeuten. Daher ist es an allen Stellen an denen vektorisierter Code und nicht vektorisierter Code aufeinander zurückgreifen notwendig, Marshalling einzusetzen, beispielsweise Datentypen ineinander umzurechnen. An diesen Stellen entsteht also ein unter Umständen großer Mehraufwand.
Für die Vektorisierung entscheidend werden diese zwei Aspekte sein: Repräsentation verschachtelter Arrays und Repräsentation vektorisierter Funktionen. Wir konzentrieren uns daher auf diese beiden Aspekte. Man stellt später fest, dass die eingeführten Konzepte tragfähig genug sind, um auch komplexere Haskell-Konstrukte wie algebraische Datenstrukturen zu handhaben. Um die nötige Flexibilität zu besitzen wird es für Arrays notwendig sein auf die in GHC implementierten assoziierten Datentypen zurückzugreifen und für vektorisierte Funktionen werden wir Closures verwenden.
Zunächst zeigen wir die grundsätzliche Idee der Vektorisierung informell und beispielhaft auf. Dazu betrachten wir die Funktion

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

Zu jeder solchen Funktion konstruiert der Vektorisierungsalgorithmus eine geliftete Funktion

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

wobei $*_L$ und $+_L$Vektoroperationen sind die parallel ausgeführt werden können. Semantisch gilt die Gleichung

$$\mathtt{f_L} = \mathtt{mapP\,f}$$

die grob gesprochen im Quellcode als Ersetzungsregel verwendet wird. Damit dieses Liften funktioniert müssen jedoch alle Funktionen im Programm geliftet werden, indem Funktionsaufrufe durch die zugehörige geliftete Funktion und Konstanten durch Aufrufe von `replicateP` ersetzt werden. Ein Problem entsteht jedoch bei verschachtelter Parallelität:

```
f :: a -> b

g :: [:[:a:]:] -> [:[:b:]:]
g xs = mapP (mapP f) xs
```

An dieser Stelle müsste $\mathtt{f_{LL}}$ konstruiert werden. Die Tiefe dieser Verschachtelung ist jedoch nicht statisch beschränkt und daher zur Compilezeit nicht bekannt, beispielsweise wenn Rekursion in Programmen verwendet wird. Die wesentliche Idee zur Lösung dieses Problems wurde von Blelloch angegeben und besteht darin, dass $\mathtt{f_{LL}}$ mit Hilfe von $\mathtt{f_L}$ ausgedrückt werden kann, indem das verschachtelte Array zunächst zu einem flachen Array umgewandelt wird, welches nur noch die Daten enthält und keine Verschachtelungsinformation mehr (`concatP`). Auf dieses flache Array lässt sich $\mathtt{f_L}$ anwenden und im Anschluss muss die Verschachtelung rekonstruiert werden (`unconcatP`):

```
f_LL :: [:[:a:]:] -> [:[:b:]:]
f_LL xss = unconcatP xss (f_L (concatP xss))
```

Damit diese Idee effizient funktioniert ist es notwendig, `concatP` und `unconcatP` in konstanter Zeit zu implementieren. Wir werden also die Repräsentation paralleler Arrays so wählen müssen, dass diese Implementierungen möglich werden, wozu es beispielsweise zwingend erforderlich ist, verschachtelte Arrays bereits flach zu speichern mit einer zusätzlichen Information über die Verschachtelungsstruktur.
Wir führen daher einen neuen Datentyp `PA a` ein, der parallele Arrays mit Elementen des Typs `a` repräsentiert. Der Vektorisierer transformiert dann alle Typen `[:a:]` in den Typ `PA a` welcher Arrays geeignet speichert.

$$\mathcal{V}_t[\![\,[:\mathtt{a}:]\,]\!] = \mathtt{PA}\,\mathcal{V}_t[\![\,\mathtt{a}\,]\!]$$

Damit die Repräsentation effizient ist müssen wir uns die Möglichkeit eröffnen, diese für jeden Typ `a` individuell wählen zu können. In Haskell werden Listen polymorph umgesetzt, d.h. es werden Zeiger auf die tatsächlichen Inhalte verwendet: Eine Liste von `Int`-Zahlen wird dargestellt durch eine generische Liste die Zeiger auf Speicherzellen mit `Int`-Zahlen enthält. Diese Indirektion ist für unsere Belange zu ineffizient. Wir wollen ein Array von `Int`-Zahlen als einen geschlossenen Block im Speicher bestehend aus 32-Bit Einheiten repräsentieren und analog ein Array von Float-Zahlen durch einen geschlossenen Block von 64-Bit Einheiten. Problematisch ist, dass damit zunächst keine polymorphen Funktionen über den Typ `PA a` mehr sinnvoll sind, beispielsweise ist

```
lengthPA :: PA a -> Int
```

keine mögliche Definition mehr, da `lengthPA` nicht allgemein für beliebiges `a` die konkrete Repräsentation von `PA a` kennt. Tatsächlich muss `lengthPA` für jeden Typ `a` eigens an dessen Realisierung angepasst implementiert werden. Daher werden assoziierte Typen verwendet. Wir definieren die Klasse aller Typen, die Element eines parallelen Arrays sein können:

```
class PAElem a where
 data PA a                                    -- !ASSOZIIERTER DATENTYP!
 indexPA :: PA a -> Int -> a
 lengthPA :: PA a -> Int
 replicatePA :: Int -> a -> PA a
 -- ... und weitere polymorphe Operationen für parallele Arrays
```

Wesentlich ist, dass wir nicht nur Definitionen für Funktionen angeben, die konkrete Instanzen dieser Klasse implementieren müssen, sondern auch die Deklaration des Datentyps `data PA a,` die ebenfalls abstrakt ist in dem Sinne, dass Instanzen dieser Klasse implementieren müssen, wie ein Objekt vom Typ `PA a` konkret aussieht. Wir haben also die Möglichkeit diese Implementierung für jede Instanz anders zu wählen und die Implementierung der abstrakten Funktionen für eine konkrete Instanz muss sich dann dieser Information bedienen. Primitive Datentypen wie `Int` werden beispielsweise alle nach dem folgenden Muster implementiert:

```
instance 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)
 -- ...
```

Hierbei greifen wir auf die von GHC bereitgestellten Primitiven `ByteArray` und Operationen darauf zurück, wobei ein `ByteArray` in GHC einfach ein Block von Bytes ist. Wir finden also die Idee wieder, Arrays von `Int`-Objekten direkt als geschlossener Speicherblock ohne Indirektion über Zeiger zu repräsentieren. Dieser Ansatz ist auch unabhängig von DPH interessant, da er die Effizienz eines Haskell-Programms erhöhen kann. Daher sind assoziierte Datentypen im GHC bereits bereits unabhängig von DPH implementiert.
Interessant ist, wie wir damit kompliziertere Typen als Elemente von parallelen Arrays implementieren können. Für Tupel von Typen `(a, b)` beispielsweise ist folgende Implementierung sinnvoll:

```
instance (PAElem a, PAElem b) => PAElem (a, b) where
 data PA (a, b) = ATup Int (PA a) (PA b)
 indexPA (ATup _ arr1 arr2) i = (indexPA arr1 i, indexPA arr2 i)
 lengthPA (Atup n _ _) = n
 -- ...
```

Wir speichern Arrays von Tupeln also, indem wir zwei Arrays mit den Komponentendaten getrennt voneinander speichern. Damit erhalten wir die Möglichkeit `zipP` und `unzipP` in konstanter Zeit zu implementieren

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

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

indem einfach die ohnehin getrennten Arrays für die Komponenten zusammengefügt bzw. separiert werden. In dem gleichen Sinn sollen nun verschachtelte Arrays vom Typ `PA (PA a)` derart implementiert werden, dass `concatP` und `unconcatP` in konstanter Zeit möglich sind. Wir nehmen dazu die Sichtweise ein, dass ein verschachteltes Array nichts anderes ist als ein flaches Array vom Typ `PA a`, welches alle Daten enthält und ein Segmentdeskriptor des Typs `PA (Int, Int)`, der die Verschachtelungsinformation enthält. Dies führt direkt auf folgende Deklaration:

```
instance PAElem a => PAElem (PA a) where
 data PA (PA a) = AArr (PA a) (PA (Int, Int))
 indexPA (AArr arr segd) i = slicePA arr (indexPA segd i)
 lengthPA (AArr _ segd) = lengthPA segd
 -- ...
```

Dabei sei bemerkt, dass man `slicePA`, welches das "Herausschneiden" eines Teilarrays umsetzt, durch geeignete Implementierung in den Instanzen von `PAElem` in konstanter Zeit erreichen kann. Der Segmentdeskriptor wird kodiert, indem für jedes Element des äußeren Arrays ein Eintrag im Deskriptor angelegt wird, welcher (erste Komponente) die Anfangsposition der Daten dieses Elements und (zweite Komponente) die Länge der Daten im flachen Datenarray enthält. Um diesen für uns zentralen Datentyp verständlicher zu machen geben wir zwei Beispiele an:

```
[:[:1,2,3:], [:4,5:], [::], [:6,7:]:] :: PA (PA Int)
```

wird repräsentiert durch das Objekt:

```
AArr [#1,2,3,4,5,6,7#]              -- Daten
     (ATup [#0,3,5,5#] [#3,2,0,2#]) -- Segmentdeskriptor
```

Analog wird das kompliziertere Array

```
[:[:(0,15),(2,9),(3,20):], [::], [:(3,46):]:] :: PA (PA (Int, Int))
```

repräsentiert durch:

```
AArr (Atup [#0,2,3,3#] [#15,9,20,46#])  -- Daten
     (Atup [#0,3,3#] [#3,0,1#])         -- Segmentdeskriptor
```

wobei das flache Datenarray hier einfach `[:(0,15), (2,9), (3,20), (3,46):] :: PA (PA (Int, Int))` repräsentiert. Damit sind nun `concatPA` und `unconcatPA` und damit `concatP` und `unconcatP` tatsächlich in konstanter Zeit möglich:

```
concatPA :: PA (PA a) -> PA a
concatPA (AArr data _) = data

unconcatPA :: PA (PA a) -> PA b -> PA (PA b)
unconcatPA (AArr _ segd) data = AArr data segd
```

Die Implementierung "vergisst" im Fall von `concatPA` einfach den Segmentdeskriptor und im Fall von `unconcatPA` fügt sie den Segmentdeskriptor aus dem ersten gegebenen Array mit der Verschachtelungsstruktur, die dem zweiten, flachen gegeben werden soll, einfach hinzu.

Als nächstes müssen wir uns überlegen, wie vektorisierte Funktionen umzusetzen sind. Man erwartet zunächst die Transformation:

$$\mathcal{V}_t[\![\mathtt{a} \to \mathtt{b}]\!] \longmapsto (\mathcal{V}_\mathtt{t}[\![\mathtt{a}]\!] \to \mathcal{V}_\mathtt{t}[\![\mathtt{b}]\!])$$

Es treten jedoch zwei Probleme auf, die diesem Vorgehen entgegenstehen.
Zunächst stehen uns in Haskell Funktionen höherer Ordnung zur Verfügung, welche zu folgender Schwierigkeit führen: Der Code

```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, mapP g [:1,2,3:])
```

wird vom Vektorisierungsalgorithmus wie bisher beschrieben transformiert zu:

```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, g_L [:1,2,3:])
```

Da hier Funktionen höherer Ordnung verwendet werden ist zur Compilezeit unbekannt, welches `g` die Funktion `f` übergeben bekommt. Da wir allein aus einem übergebenen `g :: Int -> Bool` die Funktion $\mathtt{g}_\mathtt{L}$ nicht ermitteln können müssen `f` bei Aufruf also sowohl `g` als auch $\mathtt{g}_\mathtt{L}$ übergeben werden. Hier ist bereits klar, dass wir einen neuen Typ für vektorisierte Funktionen benötigen, der sowohl `g` als auch $\mathtt{g}_\mathtt{L}$ enthält.
Es tritt aber noch eine weitere Schwierigkeit auf: Wir betrachten den Code

```
dist :: Float -> Float -> Float
dist x y = sqrt(x*x + y*y)

distance :: [:Float:] -> [:Float -> Float:]
distance xs = mapP dist xs

-- Wie soll x repräsentiert werden?
x :: [:Float -> Float:]
x = distance [:1.0, 4.5, 6.7:]
```

Problematisch ist hier, dass jedes Element in `x` eine Funktion mit einem anderen Code ist. Wenn aber garantiert werden soll, dass DPH Programme auch auf Vektorrechnern ausgeführt werden können, darf auf mehreren Recheneinheiten, auf denen parallel gerechnet wird, nur identischer Code ausgeführt werden.
Die Lösung liegt in der Feststellung, dass `x` tatsächlich immer den gleichen Code enthält, jedoch mit anderen Werten an gewisse Variablen der Funktion `dist`. `x` ist also eigentlich ein Array von `Float`-Zahlen mit einer zusätzlichen Referenz auf die Funktion `dist`. Wir haben es hier mit einem Closure zu tun: Wir schließen Werte für einen Teil der Parameter einer Funktion bereits in einer neuen Funktion mit ein.
Diese Probleme führen uns nun zur folgenden Deklaration des Typs :-> der vektorisierte Funktionen darstellen wird:

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

Der Vektorisierungsalgorithmus nimmt also folgende Transformation vor:

$$\mathcal{V}_t[\![\mathtt{a} \to \mathtt{b}]\!] \longmapsto (\mathcal{V}_\mathtt{t}[\![\mathtt{a}]\!] :\!\to \mathcal{V}_\mathtt{t}[\![\mathtt{b}]\!])$$

In diesem Datentypen enthalten ist die "skalare Version" der ursprünglichen Funktion ($\mathtt{clo}_\mathtt{S}$), die auf parallele Arrays geliftete Funktion ($\mathtt{clo}_L$) und die Bindungen von freien Parametern beider Funktionen (`env`) wobei hier als Typ jede Instanz von `PAElem` zugelassen ist. Dementsprechend finden wir diese Bindungen am Anfang von

$\texttt{clo}_S$ und $\texttt{clo}_L$ wieder. Der Vektorisierungsalgorithmus muss selbstständig aus der ursprünglichen Funktion die Funktionen $\texttt{clo}_S$ und $\texttt{clo}_L$ erzeugen. Da ein Objekt des Typs `a :-> b` nicht automatisch auf ein Objekt des Typs `a` appliziert werden kann, wird folgender Applikationsoperator bereitgestellt, der offenbar lediglich die skalare Funktion aus der vektorisierten Funktion extrahiert und die bereits gebundenen Parameter aus dem Closure einsetzt:

```
($:) :: (a :-> b) -> a -> b
($:) (Clo env fs fl) = fs env
```

Analog dazu implementieren wir einen gelifteten Applikationsoperator, der die geliftete Funktion extrahiert. Dazu müssen natürlich die Werte der bereits gebundenen Variablen entsprechend vervielfältigt werden:

```
($:_L) :: (a :-> b) -> PA a -> PA b
($:_L) (Clo env fs fl) arr = fl (replicatePA (lengthPA arr) env) arr
```

Mit diesen Deklarationen ist es nun auch möglich Arrays vektorisierter Funktionen zu repräsentieren. Dazu erklären wir die folgende Instanz der Klasse `PAElem`:

```
instance PAElem (a :-> b) where
 data PA (a :-> b) = forall e. PAElem e =>
                     AClo { aenv :: PA e
                          , aclo_S :: e -> a -> b
                          , aclo_L :: PA e -> PA a -> PA b
                          }
 indexPA (AClo env f_S f_L) i = Clo (indexPA env i) f_S f_L
 lengthPA (AClo env _ _) = lengthPA env
 replicatePA i (Clo env f_S f_L) = AClo (replicatePA i env) f_S f_L
 -- ...
```

An der Implementierung von `PA (a :-> b)` ist die Idee zur Repräsentation von Arrays von Funktionen schön erkennbar: Wir speichern nur einen Verweis auf die Funktion und ihre geliftete Variante und das Array ist durch ein Array von Werten für die bereits gebundenen Variablen gegeben, welche dazu führen, dass sich die Funktion in jedem Arrayelement entsprechend anders verhält. Interessant ist auch die Implementierung von `indexPA`: Hier wird aus dem Array ein Element extrahiert, also eine vektorisierte Funktion, indem der entsprechende Datentyp aufgebaut wird. Da das Array komplett durch die Umgebung `env` dargestellt wird, muss hierzu lediglich diesem Array der entsprechende Eintrag entnommen werden, die Einträge für die Funktionen können übernommen werden.
Ein Problem mit dem Typ `PA (a :-> b)` verbleibt jedoch: Unsere Deklaration kann nicht mit Arrays der Form

```
[:sin, cos, tan, exp:] :: Floating a => [:a -> a:]
```

umgehen, da die Funktionen im Array tatsächlich verschieden sind. Es gibt Möglichkeiten auch auf flach daten-parallelen Maschinen solche Arrays zu handhaben, wir gehen hier aber nicht weiter darauf ein.
Nachdem die wesentlichen Ideen und Hilfsmittel der Vektorisierung nun entwickelt wurden, wollen wir zunächst ein Beispiel betrachten, bevor wir (ebenfalls nur beispielhaft) anhand von `mapP` auf die Implementierung der DPH Primitiven eingehen. Wir ziehen das Beispiel einer kompletten formalen Definition des Algorithmus vor, da es uns nur um das Verständnis der Grundideen geht. Betrachtet wird die folgende Funktion:

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

Bei der Vektorisierung dieser Funktion sind keine Parameter zu binden, wir erhalten für die gebundenen Variablen im Closure der vektorisierten Funktion also den Typ Unit `()`. Daher müssen wir zunächst nachtragen wie parallele Arrays `PA ()` dieses Typs implementiert werden sollen:

```
instance PAElem () where
 data PA () = AUnit Int
 indexPA (AUnit _) i = ()
 lengthPA (AUnit n) = n
 -- ...
```

Da der Typ Unit als einziges Element das Objekt `()` enthält ist es offenbar unnötig tatsächlich ein Array für den Inhalt eines parallelen Arrays vom Typ `PA ()` anzulegen, wir müssen lediglich die Länge speichern, wie in obiger Implementierung geschehen.
Damit können wir das Resultat der Vektorisierung unserer Beispielfunktion `inc` angeben, es lautet:

```
inc_V :: Float :-> Float
inc_V = Clo () inc_S inc_L

inc_S :: () -> Float -> Float
inc_S = (\e x -> case e of
                 () -> (+)_V $: x $: 1
       )

inc_L :: PA () -> PA Float -> PA Float
inc_L = (\e x -> case e of
                 AUnit n -> (+)_V $:_L x $:_L (replicatePA n 1)
       )
```

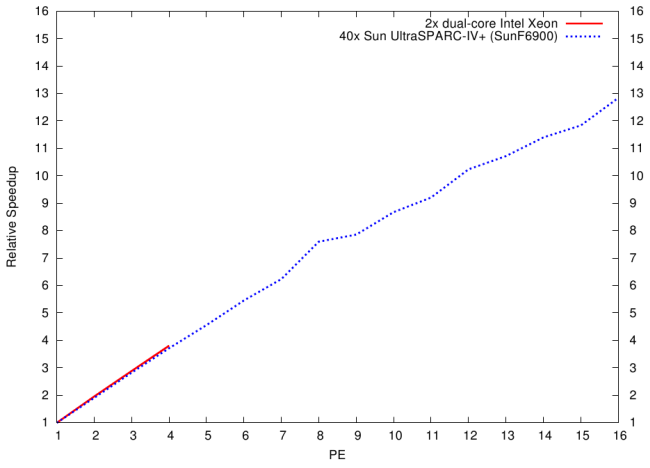
Man erkennt deutlich die Verwendung des Typs Unit für die Umgebungsvariablen, dieser muss sich nach Definition von `:->` dementsprechend auch in der skalaren und gelifteten Funktion am Anfang wiederfinden. Die `case` Ausdrücke in den erzeugten Funktionen $\texttt{inc}_S$ und $\texttt{inc}_L$ entstehen bei der automatischen Vektorisierung und sind sinnvoll, wenn es gebundene Variablen im Closure gibt. Interessant ist, dass grundsätzlich alle auftretenden Funktionen (hier: Operatoren) durch die vektorisierte Variante ersetzt wurden, z.B. + durch $+_V$. Daher wird in incS auch der skalare Applikationsoperator $: verwendet, zunächst, um + auf x anzuwenden, und anschließend um das Ergebnis auf 1 anzuwenden. In der gelifteten Version erkennt man hingegen die Verwendung des gelifteten Applikationsoperators um aus der vektorisierten Funktion die geliftete Variante zu extrahieren.
An diesem einfachen Beispiel erkennt man auch wie wichtig die weitere Programmtransformation Fusion ist: Aus einer sehr einfachen Funktion `inc` entstand durch Vektorisierung eine ziemlich komplexe Funktion in der nur für eine sehr kurze Zeit (eine Addition) tatsächlich parallel gerechnet werden kann. Außerdem erkennt man die Wichtigkeit, die dem Optimierer des Compilers zukommt: In obigem Beispiel ist der meiste Code nur aus der Möglichkeit heraus erzeugt worde, dass Funktionen höherer Ordnung auftreten. Da dies hier nicht der Fall ist wird der Optimierer des Compilers das meiste des obigen vektorisierten Codes wegoptimieren können, beispielsweise die `case` Ausdrücke. In komplexeren Programmen gewinnt dieser Schritt zunehmend an Bedeutung.
Der zweite zentrale Teil der Transformation von verschachtelter in flache Datenparallelität durch den Vektorisierungsalgorithmus ist die Implementierung der DPH Primitive. Wir stellen beispielhaft die Implementierung von `mapP` vor, in der sich die zentrale Idee von Blelloch wiederfindet. Für diese Implementierung muss offenbar eine vektorisierte Variante von `mapP` angegeben werden, also $\texttt{mapP}_V$:

```
mapP_V :: (a :-> b) :-> PA a :-> PA b
mapP_V = Clo () mapP_1 mapP_2

mapP_1 :: () -> (a :-> b) -> (PA a :-> PA b)
mapP_1 _ f = Clo f mapP_S mapP_L

mapP_2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
mapP_2 _ fs = AClo fs mapP_S mapP_L

mapP_S :: (a :-> b) -> PA a -> PA b
mapP_S (Clo env _ fl) xss = fl (replicatePA (lengthPA xss) env) xss

mapP_L :: PA (a :-> b) -> PA (PA a) -> PA (PA b)
mapP_L (AClo env _ fl) xss = unconcatPA xss (fl (expandPA xss env) (concatPA xss))
```

Blellochs Idee findet sich in den letzten beiden Zeilen in der Funktion `mapP_L` wieder: Diese Funktion wendet ein Array von vektorisierten Funktionen auf ein verschachteltes Array von Daten an. Dazu werden die Daten in ein flaches Array überführt und darauf parallel die Funktionen angewendet. Da ein Array vektorisierter Funktionen angewendet wird kann hierfür eine einzelne Funktion verwendet werden, jedoch müssen die gebundenen Variablen aus den Closures des Arrays zum verschachtelten Array passend dupliziert werden. Daher wird in den letzten Zeilen noch die Funktion `expandPA` verwendet.

Zusammenfassung

Wir haben gesehen wie es mit einer relativ kleinen Erweiterung des Sprachumfangs von Haskell möglich wird, auch komplizierte Algorithmen für eine parallele Ausführung zu implementieren. Als Beispiel haben wir den Barnes-Hut Algorithmus angegeben, der sich mit dieser Spracherweiterung bequem, knapp und elegant formulieren ließ.
Im Anschluss daran haben wir die Grundideen kennengelernt, die hinter der Implementierung von Data Parallel Haskell im GHC stehen. Hierbei sind viele Probleme offen geblieben, z.B. wie während der Vektorisierung mit algebraischen Datenstrukturen und parallelen Arrays solcher umgegangen werden muss. Komplett offen geblieben sind die beiden anderen zentralen Programmtransformationen, die DPH durchführt: Fusion und die Abbildung auf ein Programm mit expliziter Threadkontrolle.
Die Umsetzung dieser einfachen und flexibel verwendbaren Spracherweiterung stellte sich damit als schwierig heraus. Es ist daher nicht verwunderlich, dass DPH auch momentan noch nicht vollständig implementiert und die Implementierung nur schwer verwendbar ist. Dennoch gibt es von den Entwicklern vorläufige Benchmarks die zumindest belegen, dass DPH durchaus das Potential besitzt, sich zu einer guten Möglichkeit zum Formulieren paralleler Programme zu entwickeln.
Der folgende Benchmark gibt den Speedup der Multiplikation einer 10000×10000 dünn besetzten Matrix wieder, die ungefähr 1 Million Double-Einträge von Null verschieden enthält. Dabei wurde nach rechts hin die Anzahl verwendeter Recheneinheiten (PEs) aufgetragen:



Literatur

[1] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, Manuel M. T. Chakravarty
Harnessing the Multicores: Nested Data Parallelism in Haskell

[2] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, Manuel M. T. Chakravarty
Partial Vectorisation of Haskell Programs

[3] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, Manuel M. T. Chakravarty, Simon Marlow
Data Parallel Haskell: a status report

[4] Simon Peyton Jones, Gabriele Keller, Manuel M. T. Chakravarty, Simon Marlow
Associated Types with Class