Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Datum: 14.01.1997

Simplifying Parallel List Traversal

Referat für das Seminar „Parallelisierung und verteilte Ausführung funktionaler Programme“ im Wintersemester 1996/97

vorgelegt von

Markus Ueberall
(markus@informatik.uni-frankfurt.de)

Inhaltsverzeichnis

1. Einleitung

1.1. Problembeschreibung / Ziel

Im Gegensatz zu imperativen Programmiersprachen, in welchen man die Auswertung als Sequenz von Aktionen spezifiziert, wird dieselbe Aufgabe in funktionalen Sprachen als Komposition von Funktionen formuliert.

Funktionale Sprachen fördern damit einen Programmierstil, bei welchem Programme als eine Folge von Transformationen auf Datenstrukturen ausgedrückt werden. Dies erlaubt klarere und modularere Programme, trägt jedoch direkt zu den höheren Ausführungskosten (bzgl. Rechenkraft/Speicherbedarf) gegenüber imperativen Sprachen bei, da der Programmierer hierbei gezwungen ist, zusätzliche zwischengeschaltete Datenstrukturen (*intermediate data structures*) einzuführen. Da es sich bei diesen Datenstrukturen häufig um Listen handelt, wird dieser Programmierstil auch als „listendurchsetzt" (*listful*) bezeichnet.

(In Haskell gibt es eine spezielle Syntax, genannt *list comprehension*, welche eine kurze Schreibweise zum Ausdruck listenbasierter Manipulationen zur Verfügung stellt; die Präsenz solcher „Annehmlichkeiten" ist ein weiterer Beweis der Wichtigkeit von Listen in funktionalen Sprachen.)

Das Entfernen zwischengeschalteter Listen liefert zwei greifbare Vorteile:

1. Die Kosten des Konstruierens und Auflösens einer zwischengeschaltenen Datenstruktur werden vermieden; abgesehen von den Kosten der Speicherzugriffe werden hiermit auch die Speicheranforderungs- und Garbage-Collection-Mechanismen entlastet

2. zwei Komponenten eines Programms, welche zuvor durch eine zwischengeschaltete Liste voneinander getrennt wurden, „sehen" sich nun gegenseitig; das Entfernen einer zwischengeschalteten Datenstruktur könnte es anderen optimierenden Transformationen, welche die vorher vorhandene Abgrenzung nicht durchdringen konnten, erlauben, das listenlose Programm zu optimieren.

Man strebt das *automatische* Entdecken und Entfernen zwischengeschalteter Listen an, um *listfulness* (und die damit verbundene und angestrebte Übersichtlichkeit) verwenden zu können, ohne den zugehörigen Performanceverlust bezahlen zu müssen.

1.2. Ein einfaches Beispiel: Summe von Quadraten

Betrachten wir als Beispiel ein kurzes Programm, welches die Summe von Quadraten der Zahlen 1 bis n berechnet:

```
SumSq :: Int -> Int
sumSq n = sum (map square [1..n])
square :: Int -> Int
square x = x * x
```

Diese Formulierung impliziert die Verwendung zweier zwischengeschalteter Listen: die Liste mit den Zahlen 1 bis n und eine zweite Liste mit den Quadraten dieser Zahlen, welche `map square` mit `sum` verbindet.

Es ist möglich, die betreffende Funktion umzuformulieren, so daß keine solche zwischengeschalteten Listen mehr benötigt werden:

```
sumSq :: Int -> Int
sumSq n = sumSq' 1
  where
    sumSq' x = if x > n
               then 0
               else square x + sumSq' (x+1)
```

Obwohl effizienter (ein Aufruf von `sumSq 10` erfordert unter HUGS nur noch 73 statt 123 Reduktionen), ist die obige Formulierung weniger modular und schwerer zu verstehen, insbesondere weil sie nicht mehr die natürliche Pipeline-Struktur dieser Berechnung ausdrückt.

(Ein wesentlich eindrucksvolleres Beispiel, das Damenproblem (`queens`), findet sich in Anhang A.3).

1.3. Betrachtete Algorithmen-Klasse

Alle nachfolgenden Algorithmen gehören zur Klasse der Schema-basierten Systeme, welche in Programmen nach bestimmten bekannten Mustern von Funktionen suchen und diese durch Verwendung von Regeln über diese Muster optimieren. In einigen Fällen können diese Transformationen verwendet werden, um zwischengeschaltete Datenstrukturen zu eliminieren; eine solche algebraische Transformation zur Eliminierung einer zwischengeschalteten Liste ist beispielsweise die `map`-of-`map`-Regel:

```
map f (map g xs) = map (f.g) xs
```

Es ist einfach zu zeigen, daß eine große Menge von algebraischen Regeln genauso unpraktisch sein würde wie für ein allgemeines Schema zur Entfernung von Listen. Für n verschiedene listenverarbeitende (d.h. -konsumierende) Funktionen würden n^2 Regeln benötigt; außerdem kann prinzipiell jede algebraische Transformation eine völlig neue listenverarbeitende Funktion erzeugen.

Eine Möglichkeit, diese Regeln-Explosion anzugreifen, ist es, das Wesentliche der Listenmanipulationen herauszufiltern (wie die Erzeugung/Verarbeitung/Abbildung von Listen, usw.) und Listenverarbeiter (*list processors*) durch eine kleine Menge von Kombinatoren auszudrücken: Eine kleine Menge von Regeln, welche auf diesen Kombinatoren arbeiten, kann dann zwischengeschaltete Listen eliminieren.

2. Erster Ansatz: Wadlers deforestation-Algorithmus

Wadlers *deforestation*[1]-Algorithmus ist ein auf der fold/unfold-Strategie (vgl. Anhang C) basierendes Transformationssystem, welches zwischengeschaltete Datenstrukturen durch „automagisches" Anwenden der Regeln von Burstall/Darlington aus funktionalen Programmen entfernt.

(Traditionelle) deforestation läßt sich nur auf funktionale Programme erster Ordnung anwenden, welche aus Funktionen in einer bestimmten, „baumlos" genannten Form zusammengesetzt sind (Abbildung 1). Wie dieser Name vermuten läßt, garantiert eine Funktion in baumloser Form, daß sie niemals zwischengeschaltete Datenstrukturen (d.h. Bäume) generiert; eine weitere Einschränkung dieser Funktion ist, daß ihre Argumente *linear* sein müssen, d.h. kein Argument darf im Funktionsrumpf mehr als einmal auftreten (ein klassisches Beispiel einer nicht-linearen Funktion ist `square x = x*x`; eine alternative lineare Formulierung wäre `square x = exp(2*log x)`).

Die Idee ist folgende: Das Programm besteht aus Funktionen in o.g. baumloser Form und einem applikativen Term, welcher durch Variablen und Aufrufe solcher Funktionen gebildet werden kann; dieser wird immer dann zwischengeschaltete Bäume verwenden, wenn ein Funktionsaufruf (welcher einen Baum generiert) als Argument eines anderen Funktionsaufrufs auftaucht.

Jedes solche Programm kann in ein anderes übersetzt werden mit neuen, verzahnten (*mutually recursive*) Funktionsdefinitionen mit einen Rumpf in baumloser Form; dieses erzeugt keine zwischengeschalteten Bäume mehr.

Informativ arbeitet der Algorithmus wie folgt: Die Transformationsfunktion (auf die einzelnen Regeln aus (Wadler, 1990) soll an dieser Stelle nicht eingegangen werden) wird auf die ursprüngliche Funktionskomposition angewandt und entfaltet kontinuierlich Funktionsaufrufe, während Reduktions- und Substitutions-Transformationen auf den Ausdruck angewandt werden. Jedesmal, wenn ein Teilausdruck auftritt,

[1] Das Entfernen zwischengeschalteter Datenstrukturen wird traditionell „deforestation" (auf deutsch: Abholzung, Abrodung) genannt; diese Bezeichnung des Algorithmus' ist naheliegend, wenn man diese Datenstrukturen allgemein als „Bäume" ansieht.

$tt \;::=\; v$	Variable
$\quad \mid\; c\;\; tt_1 \ldots tt_k$	Konstruktor-Anwendung
$\quad \mid\; f\;\; v_1 \ldots v_k$	Funktions-Anwendung
$\quad \mid\; \textbf{case}\; v_0\; \textbf{of}\; p_1 : t_1 \mid \cdots \mid p_n : t_n$	Fallunterscheidung
$p \;::=\; c\;\; v_1 \ldots v_k$	Muster (pattern)

In einer Anwendung werden $tt_1, \ldots, tt_n$ Argumente genannt; in einer Fallunterscheidung bezeichnet man t_0 als Selektor und $p_1 : tt_1, \ldots, p_n : tt_n$ als Zweige; Funktionsdefinitionen besitzen die Form $f\;\; v_1 \ldots v_k \;=\; tt$

Abbildung 1: Definition von baumlosen Termen (*treeless terms*) in Backus-Naur-Notation

welcher eine Umbenennung eines zuvor transformierten Ausdrucks ist, wird der fold-Schritt ausgelöst. Dieser fold-Schritt generiert eine neue rekursive Funktion, wobei sichergestellt ist, daß diese in baumloser Form ist. Der Algorithmus fährt fort, bis keine weiteren Teilausdrücke mehr transformiert werden können, d.h. bis ein Ausdruck und eine neue Gruppe von baumlosen Funktionen existieren; nach Definition der baumlosen Form bedeutet dies gerade, daß das Ergebnis keine zwischengeschalteten Datenstrukturen mehr enthält.

Die Terminierung des Algorithmus' basiert lediglich auf der Tatsache, daß der fold-Schritt *immer* irgendwann einmal auftritt, da die Definition der baumlosen Form dies gewährleistet (in (Wadler, 1990) findet sich eine Beweisskizze).

Unglücklicherweise ist die o.g. baumlose Form eine sehr restriktive Sprache, sowohl in ihrer Form als auch in ihrer Ausdrucksfähigkeit; die meisten modernen funktionalen Programmiersprachen erlauben Funktionen höherer Ordnung. Die beiden anderen Einschränkungen (daß Argumente einer Funktionsanwendung nur Variablen sein können und Funktionsargumente linear sein müssen) bewirken, daß es nicht möglich ist, eine beliebige Funktion in baumloser Form zu schreiben, weil man im allgemeinen nicht immer auf zwischengeschaltete Datenstrukturen verzichten kann.

Eine attraktive Eigenschaft des deforestation-Algorithmus' ist, daß er komplett *transparent* ist. Durch eine syntaktische Restriktion der Eingabe stellt deforestation sicher, daß zwischengeschaltete Datenstrukturen immer entfernt werden.

2.1. Lockerung der Restriktionen

Um die o.g. Restriktionen zu entfernen/mildern, wurden in (Wadler, 1990) zwei Methoden vorgeschlagen:

- *blazing*[2] lockert die Einschränkung, daß Funktionen nur Variablen als Argumente besitzen dürfen, indem Ausdrücke von numerischem (oder anderen nicht-rekursivem) Typ in Argumentpositionen erlaubt werden; diese Ausdrücke können nur ein einfaches Ergebnis liefern und als solche zählen sie nicht wirklich als zwischengeschaltete Datenstrukturen. Im solcherart erweiterten *blazed deforestation*-Algorithmus werden solche „negativ" (im Sinne von: nicht zu optimierenden) markierten Ausdrücke nicht weiter transformiert.

  blazing erweitert den Spielraum von als Eingabe verwendbarer Funktionen, jedoch werden immer noch nicht alle Funktionen abgedeckt.

- eine Technik, welche „Makros höherer Ordnung" genannt wird, erlaubt die Darstellung bestimmter Funktionen mit Argumenten vom Funktionstyp als Makros dank Abstraktion der problematischen Argumente; dies ermöglicht es Funktionen höherer Ordnung wie `map`, in baumloser Form erster Ordnung dargestellt zu werden.

[2] wiederum ein (passender) Begriff aus der Forstsprache, welcher das Markieren der Baumrinden bezeichnet

Die verbleibende Einschränkung, daß alle Funktionsargumente linear sein müssen, ist vorhanden, um sicherzustellen, daß der deforestation-Algorithmus kein *weniger* effizientes Programm produzieren kann. Dies kann während der unfold-Phase geschehen, in welcher eine Funktionsanwendung durch eine Instanz der Funktionsdefinition ersetzt wird. Wenn es einem Argument erlaubt wäre, mehrere Male im Funktionsrumpf aufzutauchen, könnte als Ergebnis des unfolding eine Ausdrucks-Duplizierung auftreten, was bewirken könnte, daß eine beliebig große Berechnung zur Laufzeit dupliziert wird.

2.2. higher order deforestation

Es existiert eine neuere Erweiterung des Algorithmus', genannt *higher order deforestation* (Marlow, 1995), für welche die o.g. Restriktionen nicht gelten:

- der Algorithmus arbeitet nun auf einer Sprache höherer Ordnung
- Angaben in der Quellsprache, welche das Entfernen zwischengeschalteter Datenstrukturen steuern, sind möglich; dies scheint wesentlich angenehmer und flexibler zu sein als blazing nach Typen, welches als zu grob angesehen wird

Ein *Prototyp* dieses Transformers wurde in Haskell implementiert und ist Bestandteil des Compilers GHC.

3. Eine Alternative: Multi-Level-deforestation-Systeme

Ein völlig anderer Ansatz zur Entfernung zwischengeschalteter Datenstrukturen als die traditionelle deforestation, wobei versucht wird, eine Regel anzuwenden, welche über *einen* Level einer Datenstruktur arbeitet, und diese so zu verändern, daß sie auch auf rekursive Datenstrukturen anwendbar ist, besteht in der Entwicklung einer neuen Regel, welche über die *gesamte* Länge der Datenstruktur arbeitet:

- Anstelle einer Fallunterscheidung mittels `case`, um auszudrücken, wie die Liste zerlegt wird, wobei ein Konstruktor nach dem anderen konsumiert wird, verwendet man eine Funktion (`foldr`[3]), welche eine komplette Liste konsumiert
- Anstelle der Verwendung mehrerer individueller Konstruktor-Zellen zur Konstruktion einer Liste, ein Konstruktor nach dem anderen, verwendet man eine neue Funktion (`build`), welche die gesamte Liste konstruiert
- Wegen der einheitlichen Behandlung von Listen als komplette Objekte, anstatt sie als Kompositionen verschiedener individueller Konstruktor-Zellen anzusehen, kann *eine einzige Regel*, die `foldr`/`build`-Regel, angewendet werden, welche analog ist zur Fallunterscheidungs-Reduktionsregel (siehe Beispiel in Anhang A.1.1) aber *die komplette Datenstruktur* anstelle eines einzigen Konstruktors entfernt
- Der Nachteil dieses – bequemeren – Ansatzes besteht darin, daß der Algorithmus in dieser Form aufgrund der verwendeten Funktionen nicht auf beliebigen Datenstrukturen operieren kann (d.h. es ist nicht garantiert, daß alle zwischengeschalteten Listen entfernt werden)

3.1. Funktionsweise von cheap deforestation

Die Haskell-Definition von foldr in der Standard-Bibliothek lautet:

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

[3]nicht zu verwechseln mit der fold-Regel von Burstall/Darlington!

`foldr` beschreibt eine rekursive Listenverarbeitung, und seine Argumente geben an, wie diese Verarbeitung durchgeführt wird. Wenn `foldr` k z auf eine Liste angewandt wird, wird jede `(:)`-Zelle durch k und das abschließende `[]` durch z ersetzt (siehe Abbildung 2).

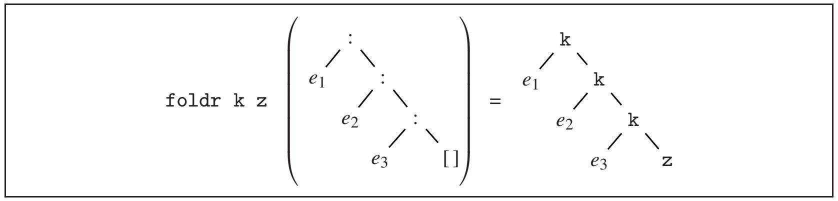

Abbildung 2: Transformation einer Liste durch `foldr`

Betrachten wir die einfache Liste von Konstanten `1 : 2 : 3 : []`. Wir können die `(:)` und `[]` durch Abstrahieren der Liste über `(:)` und `[]` herausheben (*lifting*)[4]:

```
(\ c n -> 1 ‘c‘ (2 ‘c‘ (3 ‘c‘ n))) (:) []
```

Um der Transformation zu ermöglichen, das Abstrahieren von Listen über ihre `(:)` und `[]` zu erkennen, wird die Funktion `build` verwendet, um solche Listen zu repräsentieren; ihre Definition lautet:

```
build :: ((a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

In unserem o.g. Beispiel würden wir erhalten:

```
build (\ c n -> 1 ‘c‘ (2 ‘c‘ (3 ‘c‘ n)))
```

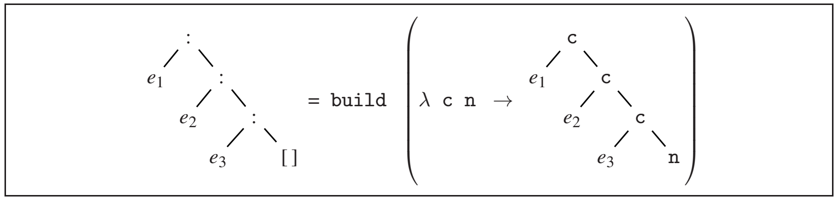

Abbildung 3: Ausdruck einer Liste durch `build`

Wenn eine Liste ohne `build` erzeugt oder ohne `foldr` verarbeitet wird, wird sie durch *cheap deforestation* nicht entfernt; solche Methoden der Listenkonstruktion und -verarbeitung stellen jedoch keine Behinderung des Algorithmus' dar.

Andererseits lassen sich eine Reihe von Funktionen durch `build`, `fold`, oder beide zusammen ausdrücken (Abbildung 4).

Wenn nun eine Liste mittels `build` konstruiert und mittels `foldr` konsumiert wird, kann die Schlüsseltransformation — die `foldr`/`build`-Regel — verwendet werden, um eine zwischengeschaltete Liste zu eliminieren (vgl. Abbildung 5):

$$\texttt{foldr}\ k\ z\ (\texttt{build}\ g)\ =\ g\ k\ z$$

[4]<*exp1*> `‘c‘` <*exp2*> ist Haskells syntaktischer Zucker für `c` <*exp1*> <*exp2*>

```
-- Listenverarbeiter
sum = foldr (+) 0 xs

-- Listenerzeuger
from x y =
    build (\ c n ->
    let
        from’ x y = if x > y
            then n
            else x ‘c‘ from’ (x+1) y
    in from’ x y)

-- Funktionen, welche Listen sowohl produzieren als auch verarbeiten
map f xs = build (\ c n -> foldr (\ a b -> f a ‘c‘ b) n xs)
concat xs = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)
xs ++ ys = build (\ c n -> foldr c (foldr c n ys) xs)
```

Abbildung 4: Einige Funktionsdefinitionen, formuliert mittels `foldr` und `build`

Angenommen, g konstruiert sein Ergebnis, indem es die durch `build` an sich als Argumente übergebene Cons- und Nil-Werte verwendet; weil `foldr k z` k für `(:)` und z für `[]` substituiert, kann der gleiche Effekt wie die Generierung und Verarbeitung einer Liste dadurch erreicht werden, daß k und z direkt an g übergeben werden. Die Gültigkeit dieser Annahme kann sichergestellt werden, indem `build` einen speziellen Typ erhält (Gill, 1996).

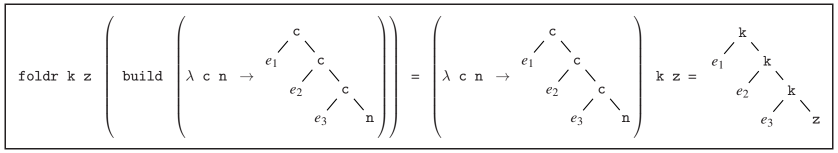

Abbildung 5: Arbeitsweise der `foldr`/`build`-Regel

4. Vergleich der beiden Ansätze

Die Beziehung zwischen traditioneller (bzw. higher order) deforestation und cheap deforestation ist (auch wenn wir nur Listen betrachten[5]) nicht offensichtlich, jedoch lassen sich prinzipielle Unterschiede anführen:

- Eine signifikante Unzulänglichkeit von cheap deforestation ist es, daß sie im Gegensatz zu higher order deforestation die Entfernung zwischengeschalteter Listen zwischen einem Verbraucher und mehreren Erzeugern nicht ausdrücken kann; der Grund hierfür liegt in der Übersetzung in eine Formulierung mittels `foldr` und Anwendung der `foldr`/`build`-Regel, wodurch nur ein einziges Argmunent optimierbar ist.

  Als klassisches Beispiel betrachten wir die `zip`-Funktion: `zip (map f xs) (map g ys)`. Durch Verwendung von `foldr` können wir die erste Liste konsumieren:

[5]higher order deforestation verarbeitet beliebige Datenstrukturen, während cheap deforestation nur über Listen definiert, u.U. jedoch erweiterbar ist (Marlow, 1995)


```
zip xs ys = foldr f (\ x -> []) xs ys
    where
        f x g []     = []
        f x g (y:ys) = (x,y) : g ys
```

  Es ist jedoch nicht möglich, gleichzeitig die zweite Liste mittels `foldr` zu konsumieren — wir können noch nicht einmal wählen, welches Argument durch diese Funktion verarbeitet werden soll. Es scheint, als ob die obige Technik auch auf das zweite Argument von `zip` anwendbar ist, aber die Definition von `zip` besitzt eine asymmetrische Semantik:

$$\texttt{zip xs ys} \not\equiv \texttt{map (\textbackslash\ (x,y) -> (x,y)) (zip ys xs)}$$

  Dies kommt daher, daß `zip` $\perp$ [ ] = $\perp$, aber `zip` [ ] $\perp$ = [ ] ergibt. (Selbst wenn `zip` eine symmetrische Semantik besitzen würde, könnte cheap deforestation zwar wählen, welches Argument durch `foldr` konsumiert werden soll; das Problem, beide Listen zu eliminieren, würde dadurch jedoch nicht gelöst!)

  Eine „Schadensbegrenzung“ in diesem Fall bestünde darin, spezielle Formulierungen von `zip` zu verwenden, wenn diese Funktion zusammen mit einem Enumerator Verwendung findet, zum Beispiel `zip [n..]` <*exp*> oder `zip` <*exp*> `[n..m]` (Gill, 1996).

- Ein anderes Beispiel, welches die Art und Weise beleuchtet, in der traditionelle deforestation erfolgreich(er) „irreguläre“ Listen-Verbraucher (d.h. Funktionen, welche einige Elemente einer Liste unterschiedlich von anderen behandeln und dadurch nur „unbequem“ mittels `foldr` ausgedrückt werden können) transformiert, ist das Entfernen von Listen innerhalb der Funktion `fourth`, welche wie folgt definiert ist:

```
second (x : _ : xs) = x : second xs
second [x] = [x]
second []  = []
fourth = second . second
```

  Traditionelle deforestation entfernt die Liste zwischen den beiden `second`-Aufrufen durch Konstruktion einer effizienten Kaskade von Fallunterscheidungen, während cheap deforestation zwei boolesche Hilfsvariablen benötigt (was damit zusammenhängt, daß die Umformulierung von `second` mittels `foldr` und `build` einen bidirektionalen Datenfluß mit sich zieht (Gill, 1996)) — dies zeigt, daß man Listen mit cheap deforestaion nicht immer entfernen sollte, nur weil dies möglich ist.

- Als Gegenbeispiel betrachten wir den Ausdruck `map f . reverse . map g`, welcher zwei zwischengeschaltete Listen (v.l.n.r.: „A“ und „B“) besitzt. Durch cheap deforestation werden *beide* Listen entfernt, obwohl B durch eine Sequenz von sog. „Aufhängern“ (suspensions) innerhalb eines akkumulierenden Arguments ersetzt wird; in einigen Fällen kann jedoch der strictness-Analyser verwendet werden, um festzustellen, daß dieses Argument strikt und der von Liste B verwendete Platz vollständig einzusparen ist (Gill, 1996).

  Bei higher order deforestation muß an der baumlosen Form von Listenerzeugern festgehalten werden: Obwohl Liste B problemlos (ohne Aufhänger) entfernt wird, erzwingt in unserem Beispiel diese baumlose Form von `reverse`, daß die erzeugte Liste A erhalten bleibt (Marlow, 1995):

```
reverse = λxs . λys case xs of
    Nil       → ys
    Cons x xs → let x = Cons x ys in reverse xs z
```

- Im Gegensatz zur traditionellen deforestation ist der Terminierungsbeweis bei cheap deforestation trivial, da rekursive Funktionen nicht entfaltet werden; stattdessen wird ein ähnlicher Effekt erreicht, indem algebraische Transformationen auf vordefinierten Funktionen durchgeführt werden.

Um die wichtigen Punkte zusammenzufassen: cheap deforestation hat Schwierigkeiten mit irregulären Listen-Verarbeitern und Funktionen, welche mehrere Listen verarbeiten, ist dafür einfach zu implementieren und vollständig transparent. higher order deforestation hat hingegen Probleme mit einigen irregulären Listen-Erzeugern, ist allgemeiner definiert (und dadurch mächtiger), aber aufwendiger in der Implementation (mehrere Regeln) und (deshalb) „weniger" transparent; zudem kann der hierdurch erzeugte Code viel größer sein als nötig in Fällen, bei welchen mehrere komplexe Funktionen kombiniert werden (in (Marlow, 1995) findet sich hierzu als Beispiel das Programm `life`). Beide Ansätze terminieren garantiert, arbeiten *source-to-source* und erreichen nur einen konstanten Beschleunigungsfaktor (dies ist allgemeingültig für deforestation; dieser Faktor kann jedoch groß genug sein, daß die Optimierung signifikante Effizienzsteigerungen liefert).

5. Ein deforestation-Algorithmus für parallele funktionale Programme

Eine direkte Übertragung von cheap deforestation in eine parallele funktionale Sprache wie pH ist leider nicht möglich, da dieser Algorithmus wie gesehen auf der `foldr`-Funktion aufbaut, was die Verknüpfung von Ergebnissen in einer bestimmten Reihenfolge erzwingt. Dies jedoch stellt in pH ein ernstes Problem dar, weil hierdurch wichtige Optimierungen der Art `sum (a ++ b) = sum a + sum b` ausgegrenzt werden (in einer parallelen Sprache ermöglicht diese Umschreibung die parallele Berechnung der Ausdrücke `sum a` und `sum b`); gerade diese Optimierungen hätten jedoch eine unmittelbare Auswirkung auf list/array comprehensions, abgesehen davon, daß sie gewöhnliche Listenoperationen verbilligen würden.

Will man zumindest den Ansatz von cheap deforestation beibehalten, so muß eine Funktion gefunden werden, welche die Aufgabe von `foldr` übernehmen kann, gleichzeitig jedoch keine Anforderung an die Auswertungsreihenfolge von Listenelementen stellt, sofern diese nicht aus einem anderen Grund eingehalten werden muß.

5.1. Listendurchläufe

Betrachten wir zuerst einmal die Eigenschaften der beiden in funktionalen Sprachen üblicherweise verwendeten Funktionen zum Durchlauf von Listen: `foldr` und `foldl`. Ihre mathematischen Definitionen lauten wie folgt:

$$\texttt{foldr}\ \oplus\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots \oplus (x_n \oplus z)))$$
$$\texttt{foldl}\ \oplus\ z\ [x_1, x_2, \ldots, x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n$$

Die Klammerung zeigt an, daß `foldr` eine Liste „von rechts nach links" durchläuft, während dies bei `foldl` gerade in entgegengesetzter Richtung geschieht.

Greifen wir obiges Beispiel der Addition von Listenelementen auf: `sum` kann entweder durch `foldr (+) 0 xs` oder durch `foldl (+) 0 xs` definiert werden; die Verknüpfung $\oplus$ verbindet das „Zwischenergebnis" (zu Beginn z) mit allen Werten der x_i.

`foldl` läuft „mit den Zeigern"; das Ergebnis ist verfügbar, sobald das Listenende erreicht wurde, so daß diese Funktion endrekursiv (iterativ) definiert werden kann:

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

`foldr` hingegen läuft „gegen die Zeiger", d.h. das Ergebnis ist nicht verfügbar, bevor nicht die Liste vollständig abgearbeitet wurde — die eigentliche Berechnung findet praktisch auf dem Rückweg (rekursiv) statt (die Haskell-Definition von `foldr` findet sich in Kapitel 3.1).

Beide `fold`-Operationen erlegen dem Datenfluß in den durch sie repräsentierten Berechnungen eine bestimmte Ordnung auf, was zu einen Flaschenhals führen kann, welcher algebraisch am besten durch die folgenden Identitäten auszudrücken ist:

```
foldr f z (a ++ b) = foldr f (foldr f z b) a
foldl f z (a ++ b) = foldl f (foldl f z a) b
```

Diese Identitäten geben an, daß, sofern eine Liste in zwei Teillisten a und b aufteilbar ist, diese durchlaufen werden kann, indem beide Teillisten durchlaufen werden und das Ergebnis der ersten `fold`-Operation (als neues Zwischenergebnis) an die zweite übergeben wird; aus den obigen mathematischen Definitionen beider Operationen ergibt sich

$$\texttt{foldr} \oplus z\,[a_1, a_2, \ldots, a_n]\ ++\ [b_1, b_2, \ldots, b_n] = a_1 \oplus (a_2 \oplus (\cdots a_n \oplus (b_1 \oplus (b_2 \oplus (\cdots b_n \oplus z)))))$$
$$= \texttt{foldr} \oplus (\texttt{foldr} \oplus z\,[b_1, b_2, \ldots, b_n])\,[a_1, a_2, \ldots, a_n]$$
$$\texttt{foldl} \oplus z\,[a_1, a_2, \ldots, a_n]\ ++\ [b_1, b_2, \ldots, b_n] = (((((z \oplus a_1) \oplus a_2) \oplus \cdots a_n) \oplus b_1) \oplus b_2) \oplus \cdots b_n$$
$$= \texttt{foldl} \oplus (\texttt{foldl} \oplus z\,[a_1, a_2, \ldots, a_n])\,[b_1, b_2, \ldots, b_n]$$

Keine dieser Definitionen erinnert an die Tatsache, daß `sum (a++b)` äquivalent ist zu `sum a + sum b`. Die Lösung dieses Problems besteht darin, das Durchlaufen von Listen durch einen Operator symmetrisch zu beschreiben (`traversal (a++b) = traversal a ‘op‘ traversal b`); diese Operation wird *reduzieren* genannt und ein solcher allgemeiner Operator, `reduce`, besitzt die folgende Definition:

$$\texttt{reduce}\ \oplus\ z\,[x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$
$$\texttt{reduce}\ \oplus\ z\,[\,] = z$$

Der offensichtliche Unterschied zwischen den Definitionen von `foldl`/`foldr` ist das Fehlen der Klammern und die Tatsache, daß z nur dann explizit verwendet wird, um den Wert der Reduktion für die leere Liste zu definieren. Damit `reduce` sicher ist, müssen $\oplus$ und z (für beliebige Werte von a, b, c des passenden Typs) die folgenden Eigenschaften erfüllen:

$$a \oplus (b \oplus c) \ \equiv\ (a \oplus b) \oplus c \qquad \Rightarrow \oplus \text{ ist assoziativ}$$
$$a \oplus z \ \equiv\ z \oplus a \ \equiv\ a \qquad \Rightarrow z \text{ ist Identität von } \oplus$$

Mit anderen Worten: $\oplus$ erfüllt die Anforderungen für die binäre Funktion eines Monoids mit einer Identität; somit können wir `reduce` wie folgt umformulieren:

$$\texttt{reduce}\ (\oplus)\ z\ (a\ ++\ b) = \texttt{reduce}\ (\oplus)\ z\ a\ \oplus\ \texttt{reduce}\ (\oplus)\ z\ b$$
$$\texttt{reduce}\ (\oplus)\ z\ [e] = e$$
$$\texttt{reduce}\ (\oplus)\ z\ [\,] = z$$

`reduce` ist also eine Möglichkeit, eine Liste in zwei beliebige Teillisten aufzuteilen und parallel zu verarbeiten, wobei eine Überbestimmung vermieden wird (wir legen ja insbesondere Wert darauf, Algorithmen so zu formulieren, daß keine unechten Abhängigkeiten eingeführt werden, welche eine sequentielle Ausführung erzwingen). Schreiben wir nun `sum = reduce (+) 0`, erhalten wir das gewünschte Ergebnis:

```
sum (a++b) = reduce (+) 0 (a++b) = reduce (+) 0 a + reduce (+) 0 b = sum a + sum b
```

5.2. Homomorphismen

Unglücklicherweise ist es mit `reduce` allen nicht möglich, die folgende Idee auszudrücken:

```
length (a ++ b) = (length a) + (length b)
```

Das Problem ist offensichtlich, wenn wir die Typen der Funktionen untersuchen:

```
reduce :: (a -> a -> a) -> a -> [a] -> a
length :: [a] -> Int
```

Es müßte eine Möglichkeit geben, individuelle Elemente der Liste in Objekte des korrekten Typs zu transformieren, welche dann mit `reduce` kombiniert werden können. Die Funktion, welche dies realisiert, ist `map`:

```
map f (a ++ b) = map f a ++ map f b
map f [a]      = [f e]
map f []       = []
```

`length` kann somit wie folgt ausgedrückt werden (`const` ist ein konstanter Kombinator, welcher das erste von zwei Argumenten zurückliefert):

```
length (a ++ b) = sum . map (const 1)
```

Hierdurch wird `length` einfach in zwei Teile zerlegt: ein Teil ersetzt jedes Listenelement durch die Zahl Eins (Abbildung), der zweite summiert die resultierende Liste (Reduktion).

`map` und `reduce` können auf diese Weise verwendet werden, um den Homomorphismus `h` vom Monoid auf Listen über α, $([\alpha], ++, [], (:[]))$, auf ein beliebiges Monoid $(\beta, \oplus, id_\oplus, i)$ auszudrücken:

$\texttt{h}\ l\ \texttt{= reduce}\ \oplus\ \texttt{id}_\oplus\ \texttt{(map}\ i\ l\texttt{)}$

Im wesentlichen wird jede append-Operation `(++)` durch $\oplus$, jede leere Liste `[]` durch $\texttt{id}_\oplus$ und jede Einheits-Operation `(:[])` durch i ersetzt. Diese unbedingte Ersetzung führt zu den *promotion*-Regeln für Listen, welche Aufrufe von `map` und `reduce` durch `concat` nach innen schieben:

$\texttt{map g (concat}\ L\texttt{) = concat (map (map g)}\ L\texttt{)}$
$\texttt{reduce}\ \oplus\ \texttt{z (concat}\ L\texttt{) = reduce}\ \oplus\ \texttt{z (map (reduce}\ \oplus\ \texttt{z)}\ L\texttt{)}$

Nach Definition eines Homomorphismus' mittels `map` und `reduce` wäre es möglich, letztere Funktion direkt in `fold`-Operationen zu übersetzen (dies ist zulässig, da z Identität von f ist) — dies widerspräche jedoch dem Sinn, `reduce` in eine parallele Sprache einzuführen; stattdessen wäre es wünschenswert, alle drei Methoden des Listendurchlaufs einheitlich anwenden zu können.

Der Trick, um dies zu erreichen, besteht in der Verwendung von Funktionen höherer Ordnung. Die Vorgehensweise läßt sich in drei Schritte einteilen:

1. Zusammenfassung der Berechnung, welche mit einem Listenelement durchgeführt wird (sobald dieses Element berechnet werden kann) und dem Element selbst zu einer Einheit; dieses transformiert jedes Listenelement in eine Funktion (*Abbildung*)

2. Verwendung einer Funktion höherer Ordnung, um die neu entstandenen Funktionen zu verbinden (*Reduktion*)

3. Aufruf der resultierenden Funktion mit dem Ausgangswert des Durchlaufs (*eigentliche Berechnung*)

Die sich hieraus ergebenen Definitionen sehen aus wie folgt (`id` ist die Identität beider Monoide und liefert für eine leere Teilliste den gleichen Typ wie den übergebenen zurück, `flip` vertauscht die Reihenfolge der ersten beiden Argumente der gegebenen Funktion):

```
foldr f z xs = reduce (.) id (map f xs) z
foldl f z xs = reduce (flip (.)) id (map (flip f) xs) z
```

Eine ähnliche Formulierung existiert auch für `reduce`:

$\texttt{reduce}\ \oplus\ \texttt{z xs = reduce (\textbackslash\ r t -> (l z)}\ \oplus\ \texttt{(r z)) id (map const xs) z}$

Diese Definition leitet den Wert von z über die gesamte Reduktion weiter; er wird nur verwendet, wenn eine leere Liste angetroffen wird, ansonsten wird er verworfen.

Dieser Ansatz ist sauber — ein Listendurchlauf wird als Monoid mit der Identitätsfunktion als neuer Identität dargestellt und der Homomorphismus des Monoids wird auf ein einziges Argument angewendet, um die eigentliche Berechnung zu starten. Der *desugarer* verwendet die obigen Definitionen, um alle Verarbeitungen von Listen in einer kanonischen Form auszudrücken:

$\texttt{(reduce}\ a\ \texttt{id (map}\ u\ L\texttt{))}\ t$

Hierbei ist a eine Funktion mit mindestens drei, u eine Funktion mit mindestens zwei Argumenten und t repräsentiert entweder den Ausgangswert einer `fold`-Operation oder die Identität einer Reduktion (für `foldr f z` ist beispielsweise a = `(.)`, u = `f` und t = `z`).

Dies erlaubt es dem Optimierer, Listendurchläufe einheitlich und kompakt darzustellen; allerdings scheint es momentan so, als ob durch diesen Ansatz Aufrufe von `reduce` in Schleifen oder, schlimmer noch, rekursive Funktionsaufrufe transformiert werden (obwohl mit solchen Reduktionen gerade dieses Verhalten nach Möglichkeit vermieden werden soll).

5.3. Systematische Erzeugung von Listen

Um Listen vollständig zu entfernen und nicht nur ihren Durchlauf zu optimieren, muß die Listenerzeugung genauso systematisch wie ihre Verarbeitung beschrieben werden. Eine Teillösung stellen hierbei list comprehensions dar, da sie die Erzeugung neuer Listen mittels Durchläufen bereits existierender Listen beschreiben; es gibt jedoch viele Fälle, in welchen neue Listen ohne Bezugnahme auf andere Listen erzeugt werden, beispielsweise bei Verwendung von Haskells spezieller Syntax für arithmetische Folgen, welche als Listen ausgedrückt werden: $\texttt{[}k_1\texttt{,}\ k_2\ \texttt{..}\ k_n\texttt{]}$. Außerdem kann eine teilweise erzeugte Liste während ihrer „Fertigstellung“ bereits durchlaufen werden, was ihre Eliminierung ausschließt (ansonsten müßte sie für jeden neuen Durchlauf erneut erzeugt werden!).

Wir können Funktionen definieren, welche (endliche) Listen systematisch — ein Listenelement nach dem anderen — generieren und ein Prädikat verwenden, um die Generierung zu beenden. Eine solche Funktion ist `unfold`:

```
unfold p f v = h v
    where h v | p v       = []
              | otherwise = a:h b
                    where (a,b) = f v
```

Ihre Struktur ähnelt stark dem Aufbau der `fold`-Operationen: Ein Wert `v` wird von links nach rechts über die gesamte Listenlänge weitergereicht; nimmt dieses Prädikat den Wert *wahr* an, so ist das Listenende erreicht, ansonsten wird Funktion `f` verwendet, um ein neues Listenelement und einen neuen Wert für das Prädikat zu erzeugen (die Liste wird rekursiv erzeugt — trotzdem werden die Elemente am Listenanfang konzeptionell vor den ihnen folgenden Elementen berechnet).

Der Hauptvorteil dieser Funktion ist, daß diese Struktur regulär genug ist, um Heuristiken anwenden zu können, welche unterschiedlichen Code in Abhängigkeit von der Art der Listenverarbeitung erzeugen.

5.4. Optimierung durch Kombination beider systematischen Ansätze

Nachdem die Durchläufe in eine kanonische Form überführbar sind und Abbildungen und Reduktionen die zwischengeschalteten Listen in `map` und `concat` eliminiert werden können, müssen diese Ansätze miteinander kombiniert werden, um eine einfache Menge von Optimierungen zu erhalten.

Diese Regeln (siehe Abbildung 7) werden als `map/reduce`-Regeln bezeichnet, da die letzten drei Argumente von $\mathcal{LF}$ eine kanonische Reduktion repräsentieren, welche mittels `map` und `reduce` auf die Ausdrücke in den doppelten Klammern angewendet wird. Die `bothways`-Operation stellt eine Markierungsfunktion dar, welche auf den Wert von a angewendet wird, um die Tatsache auszudrücken, daß a keine

$$
\begin{array}{lcl}
\mathcal{TE}\llbracket[E \mid Q]\rrbracket & = & \mathcal{TF}\llbracket[E \mid Q]\rrbracket \\
\mathcal{TF}\llbracket[E \mid\,]\rrbracket & = & [\mathcal{TE}\llbracket E\rrbracket] \\
\mathcal{TF}\llbracket[E \mid B,\ Q]\rrbracket & = & \texttt{if}\ \mathcal{TE}\llbracket B\rrbracket\ \texttt{then}\ \mathcal{TF}\llbracket[E \mid Q]\rrbracket\ \texttt{else []} \\
\mathcal{TF}\llbracket[E \mid P \leftarrow L,\ Q]\rrbracket & = & \texttt{concat (map f}\ \mathcal{TE}\llbracket L\rrbracket\texttt{)} \\
 & & \quad \texttt{where f}\ P\ =\ \mathcal{TF}\llbracket[E \mid Q]\rrbracket \\
 & & \qquad\quad \texttt{f \_ = []}
\end{array}
$$

Abbildung 6: Regeln zum Entzuckern (*desugaring*) von Listen durch `map` und `concat`

$$
\begin{array}{llcl}
\mathcal{LE}\llbracket[E_1, E_2, \ldots, E_n]\rrbracket & & = & \mathcal{LF}\llbracket[E_1, E_2, \ldots, E_n]\rrbracket\ a_{list}\ u_{list}\ t_{list} \\
\mathcal{LE}\llbracket E_1 \texttt{ ++ } E_2\rrbracket & & = & \mathcal{LF}\llbracket E_1 \texttt{ ++ } E_2\rrbracket\ a_{list}\ u_{list}\ t_{list} \\
 & & & \text{usw. für andere Listenausdrücke} \\
\mathcal{LE}\llbracket\texttt{reduce}\ f\ t\ L\rrbracket & & = & \mathcal{LE}\llbracket\texttt{let z =}\ t\ \texttt{in}\ \mathcal{LF}\llbracket L\rrbracket\ (\lambda l\ r\ t.f\ (l\ z)(r\ z))\ \texttt{const t}\rrbracket \\
\mathcal{LE}\llbracket\texttt{foldr}\ f\ t\ L\rrbracket & & = & \mathcal{LE}\llbracket\mathcal{LF}\llbracket L\rrbracket\ \texttt{(.)}\ f\ t\rrbracket \\
\mathcal{LE}\llbracket\texttt{foldl}\ f\ t\ L\rrbracket & & = & \mathcal{LE}\llbracket\mathcal{LF}\llbracket L\rrbracket\ \texttt{(flip (.))}\ \texttt{(flip}\ f\texttt{)}\ t\rrbracket \\
 & & & \\
\mathcal{LF}\llbracket[E_1, E_2, \ldots, E_n]\rrbracket & a\ u\ t & = & ((u\ E_1)\ \text{`}a\text{`}\ (u\ E_2)\ \text{`}a\text{`}\ \ldots\ \text{`}a\text{`}\ (u\ E_n))\ t \\
\mathcal{LF}\llbracket E_1 \texttt{ ++ } E_2\rrbracket & a\ u\ t & = & (\mathcal{LF}\llbracket E_1\rrbracket\ a\ u\ \text{`}a\text{`}\ \mathcal{LF}\llbracket E_2\rrbracket\ a\ u)\ t \\
\mathcal{LF}\llbracket[\,]\rrbracket & a\ u\ t & = & t \\
\mathcal{LF}\llbracket A : L\rrbracket & a\ u\ t & = & (u\ A\ \text{`}a\text{`}\ \mathcal{LF}\llbracket L\rrbracket\ a\ u)\ t \\
\mathcal{LF}\llbracket\texttt{map}\ f\ L\rrbracket & a\ u\ t & = & \mathcal{LF}\llbracket L\rrbracket\ a\ (u.f)\ t \\
\mathcal{LF}\llbracket\texttt{concat}\ L\rrbracket & a\ u\ t & = & \mathcal{LF}\llbracket L\rrbracket\ a\ (\lambda l.\mathcal{LF}\llbracket l\rrbracket\ a\ u)\ t \\
\mathcal{LF}\llbracket\texttt{reverse}\ L\rrbracket & a\ u\ t & = & \mathcal{LF}\llbracket L\rrbracket\ \texttt{(swap}\ a\texttt{)}\ u\ t \\
\mathcal{LF}\llbracket\texttt{someOrder}\ L\rrbracket & a\ u\ t & = & \mathcal{LF}\llbracket L\rrbracket\ \texttt{(bothways}\ a\texttt{)}\ u\ t
\end{array}
$$

Abbildung 7: Die `map`/`reduce`-Optimierungsregeln

bestimmte Anordnung erfüllen muß und wird in denjenigen Regeln ausgenutzt, welche den Code erzeugen, mit dem die Reduktion tatsächlich durchgeführt wird.

Diese Struktur erlaubt die Eliminierung von reduzierbaren Ausdrücken zur Zeit der Übersetzung, welche ansonsten übernommen würden, wenn die Optimierung die *Promotion*-Regeln direkt verwendete. Insbesondere werden alle Aufrufe von a zur Zeit der Übersetzung durchgeführt — dies stellt kein Problem dar, da a eine reguläre Struktur besitzt (eine Gefahr gibt es: t könnte dupliziert werden — allerdings nur, wenn a durch eine Reduktion erzeugt wird, und in diesem Fall erhält t einen konkreten Namen (s.u.)).

Die Regeln sind so strukturiert, so daß sie mit Verschachtelungen umgehen können, d.h. innere Teilausdrücke werden solange nicht optimiert, bis Reduktionen die umschließenden Ausdrücke eliminiert haben, was die Ausführung der `map`/`reduce`-Regel in einem einzigen Durchlauf erlaubt.

Funktionale Argumente (welche in ihren desugaring-Regeln als f markiert sind) könnten wiederholt re-optimiert werden, wenn sie wiederholt angewendet werden. Momentan wird eine relativ einfache Heuristik verwendet, um dieses Verhalten zu kontrollieren: Wenn eine Funktion an einen Namen gebunden und danach in einem Durchlauf verwendet wird, erfolgt eine einmalige separate Optimierung; sie wird nicht von weiteren `map`/`reduce`-Optimierungen profitieren, wenn sie eine verschachtelte Liste zurückliefert. Wenn diese Funktion andererseits mit einem Lambda-Ausdruck spezifiziert wird, erfolgt eine Duplizierung und Einsetzung „bis zum geht-nicht-mehr“.

Die Optimierungsregeln spezifizieren einen Neu-Durchlauf explizit nur in drei Fällen, und zwar für die Listenverarbeiter `foldl`, `foldr` und `reduce`; in den übrigen Fällen wird eine Listen-Syntax generiert. Diese Syntax würde dann re-optimiert, bis eine zweite Menge von Namen für die bereits optimierten Listenoperationen eingeführt wird. Es ist jedoch möglich, den Neu-Durchlauf dieser Operationen vollständig

$$
\begin{array}{llcl}
\mathcal{LF}\llbracket\texttt{unfold}\ P\ F\ V\rrbracket & a\ u\ t & = & \texttt{let h v acc |}\ \mathcal{LE}\llbracket P\rrbracket\ =\ \texttt{acc} \\
 & & & \qquad\qquad\quad \texttt{| otherwise}\ =\ \mathcal{LD}\llbracket\texttt{h v'}\rrbracket\llbracket\texttt{acc}\rrbracket\ a\ (u\ e) \\
 & & & \qquad\qquad\qquad\quad \texttt{where (e,v')}\ =\ \mathcal{LE}\llbracket F\rrbracket\ \texttt{v} \\
 & & & \texttt{in h}\ \mathcal{LE}\llbracket V\rrbracket\ t \\
 & & & \qquad\qquad \mathcal{LE}\ \text{wird ausgelassen, wenn}\ u \neq u_{list} \\
\mathcal{LF}\llbracket E\rrbracket & a\ u_{list}\ t & = & \mathcal{LE}\llbracket E\rrbracket\ \texttt{++}\ t \\
\text{Ansonsten:} & & & \\
\mathcal{LF}\llbracket E\rrbracket & a\ u\ t & = & \texttt{let h [] acc = acc} \\
 & & & \qquad \texttt{h (e:es) acc =}\ \mathcal{LD}\llbracket\texttt{h es}\rrbracket\llbracket\texttt{acc}\rrbracket\ a\ (u\ \texttt{e}) \\
 & & & \texttt{in h}\ \mathcal{LE}\llbracket E\rrbracket\ t
\end{array}
$$

Wenn a von einer Reduktion stammt oder gleich `bothways` ist:

$$
\begin{array}{lcll}
\mathcal{LD}\llbracket R\rrbracket\llbracket A\rrbracket\ a\ v & = & R\ (v\ A) & a\ =\ \texttt{bothways (.)},\ v\ \text{strikt} \\
\mathcal{LD}\llbracket R\rrbracket\llbracket A\rrbracket\ a\ v & = & R\ (a\ (\lambda x.A)\ v\ A) & a\ \text{strikt im zweiten Argument} \\
\mathcal{LD}\llbracket R\rrbracket\llbracket A\rrbracket\ a\ v & = & R\ (a\ v\ (\lambda x.A)\ A) & a\ =\ \texttt{bothways}\ a',\ a\ \text{strikt im ersten Argument}
\end{array}
$$

Wenn keine der obigen Regeln anwendbar ist:

$$
\mathcal{LD}\llbracket R\rrbracket\llbracket A\rrbracket\ a\ v\ =\ a\ v\ (R)\ A
$$

Abbildung 8: Regeln, um eine existierende Liste oder `unfold` zu durchlaufen

zu verhindern, indem der innere Code vorsichtig durchlaufen wird, wenn das Ergebnis produziert wird; dies führt zu der folgenden Definition:

$$
a_{list}\ =\ \texttt{(.)},\quad u_{list}\ =\ (\lambda e\ t.\mathcal{LE}\llbracket e\rrbracket : t),\quad t_{list}\ =\ \texttt{[]}
$$

Bis jetzt wurde kein formaler Beweis für die Terminierung dieser Optimierungsstruktur durchgeführt: Informell entdeckt $\mathcal{LE}$ immmer entweder keine oder eine tatsächlich zu konstruierende Liste und fährt eine Ebene tiefer fort (ggf. für jedes dieser Listenelemente) oder entdeckt einen Listendurchlauf und eliminiert ihn, wobei eine Ebene höher in der Listenverschachtelung fortgefahren wird (in Ordnung, da mit endlichen Typen operiert wird).

Die in einem der obigen Abschnitte geäußerten Bedenken hinsichtlich der Effizienz können glücklicherweise durch die nach den angegebenen Regeln durchgeführten Reduktionen verkleinert werden; durch die in Abbildung 8 angegebenen ersten drei Regeln wird akzeptabler Code erzeugt.

Diese Definitionen wurden erhalten durch die Beobachtung, daß Listendurchläufe, wie immer, sequentiell stattfinden müssen entlang der vorgegebenen Liste und daß `unfold` entsprechend darauf warten muß, daß Daten von einer Iteration an die nächste weitergereicht werden. Dann wird der folgende sequentielle Durchlauf verwendet:

```
(\ lst -> foldr (a.u) id lst l)
```

Durch Erweitern der obigen Definition von `foldr` wird für den Lambda-Rumpf der folgende Code erhalten:

```
let h [] = id
    h (e:es) = (a (u e) (h es))
in h lst t
```

Die wichtige Beobachtung, welche uns zu den o.g. drei Regeln führt, ist, daß beide Fallunterscheidungen in der Definition in `h` eine Funktion zurückliefern; da a eine feste Definition besitzt, kann hier η-Abstraktion angewendet werden, d.h. bis auf pattern matching wird keine Berechnung zur Laufzeit nötig sein, um eine

Funktion zu erhalten, welche durch `h` zurückgeliefert wird (dieser η-Abstraktions-Schritt liefert die o.g. Erweiterung).

Die Übersetzung für `unfold` folgt mit der Beobachtung der strukturellen Ähnlichkeit zwischen Durchlauf und Generierung.

5.4.1. Ausnutzen der regulären Struktur

Die Tatsache, daß die Struktur von a bekannt ist, erlaubt die Ableitung einer Anzahl wichtiger Heuristiken, welche besseren Code erzeugen als die naive Übersetung, wie sie oben angegeben ist. Die einzigen Stellen in der obigen Funktion `h`, welche variiert werden können, sind der Wert von t (welcher außerhalb von `h` selbst liegt) und der Wert der zweiten Klausel in der Definition von `h`. Es gibt zwei grundlegende Formen, welche die zweite Klausel annehmen kann (f, g sind beliebige Funktionen):

f `(h xs acc)`	echt-rekursiv
`h xs (`*g* `acc)`	iterativ

Wenn eine Wahlmöglichkeit besteht, muß die Entscheidung auf Basis der Striktheit des Ausdrucks g basieren: Wenn g strikt ist, sollte die iterative Form gewählt werden; dies erlaubt die Generierung einer Schleife, welche in konstanter Zeit im resultierenden Code für `h` ausgeführt werden kann. Anderenfalls sollte die echt-rekursive Form verwendet werden; damit kann `h` als Ganzes einen Wert zurückliefern, bevor der rekursive Aufruf abgearbeitet wurde, und eine lazy-Sprache kann den rekursiven Aufruf vollständig vermeiden. Die Regeln in Abbildung 8 generieren die iterative Form, wann immer dies möglich ist.

Es ist zu beachten, daß die Optimierungsregel für `concat` aus Abbildung 7 angibt, daß die Argumente von u verwendet werden, um eine rekursive Optimierung durchzuführen. Wie können mit dieser Tatsache die Striktheits-Eigenschaften einer solchen Funktion ermittelt werden?

Das zweite Argument von u wird in diesem Fall direkt an die rekursive Optimierung übergeben und erscheint dort als t-Parameter; dieser Parameter wird niemals selbst optimiert (sofern der Code nicht später durch $\mathcal{LE}$ von Grund auf optimiert wird). Als Ergebnis kann die rekursive Optimierung mit einem Hilfsargument d an zweiter Stelle durchgeführt und der resultierende (konkrete) Code in bezug auf d ausgewertet werden. Jeder weitere Aufruf von v in $\mathcal{LD}$ kann sein Argument einfach an d binden und den bereits erzeugten Code verwenden. Somit werden Listen, welche bereits auf dem Heap existieren, nicht explizit auf dem Heap neu erstellt; stattdessen werden sie an die Standard-append-Funktion `(++)` übergeben. Wenn t `= []` ist, kann die Umschreibung optimiert werden in ein einzelnes $\mathcal{LE}[\![E]\!]$ (was bedeutet, daß E nicht kopiert wird). Insbesondere bedeutet diese Optimierung, daß der Ausdruck `a ++ b` nicht zu `a ++ (b ++ [])` optimiert wird.

5.5. Lifting

Es gibt viele assoziative Operationen, welche nützlich für die Listen-Operation sein könnten, jedoch keine Identität besitzen. Für diese Funktionen gibt es eine Operation namens `reduce1`, welche sie explizit *liftet* und somit für eine normale Reduktion verwendbar macht (eine ähnliche Technik existiert für `foldl1` und `foldr1`):

```
data Maybe a = Nothing | Just a
reduce1 :: (a -> a -> a) -> [a] -> a
reduce1 f = unlift . reduce (lift f) Nothing
    where
        unlift (Just x) = x
        lift f Nothing y         = y
        lift f x Nothing         = x
        lift f (Just x) (Just y) = f x y
```

Diese Definitionen verhindern das explizite Destrukturieren der zu durchlaufenden Liste, führen jedoch zusätzliche Datenstrukturen ein.

5.6. Ausnutzung von (fehlenden) Anordnungsregeln

Die `map`-Operation ist unabhängig von der Anordnung ihrer Eingabe; eine Reduktion beinhaltet jedoch eine schwache Anordnungs-Information, da die an `reduce` übergebene Operation u.U. nicht kommutativ ist. Das auffallendste Beispiel ist die Funktion `concat`: Bei einer Verschmelzung von Teillisten mittels `concat [[1,2],[3],[4,5]]` ist nicht vorhersehbar, welche beiden Listen zuerst konkateniert werden (das Ergebnis `[1,2,3,4,5]` ist jedoch bekannt).

Die Unabhängigkeit von Anordnungen bei `map` und die Anordnungsabhängigkeit von `reduce` können verwendet werden, um die folgenden Regeln zu erhalten:

`map f (reverse` L`) = reverse (map f` L`)`
`reduce` $\oplus$ `z (reverse` L`) = reduce (flip` $\oplus$`) z` L

Häufig ist eine zu reduzierende Operation jedoch entweder kommutativ, oder die betrachteten Listen werden nur als „Behältnisse“ für Daten verwendet, wobei es nicht auf die Anordnung der einzelnen Listenelemente ankommt. In diesen Fällen ist es dem Optimierer freigestellt, die Reihenfolge der Elemente zu ändern, damit Teillisten während der Berechnung am effizientesten durchlaufen werden können (beispielsweise sei angenommen, die Liste `[1..3]` könne am effizientesten vorwärts durchlaufen werden; in diesem Fall ergäbe der Ausdruck `([1..3] ++ reverse [1..3])` die Liste `[1,2,3,1,2,3]`).

5.7. Eine Bemerkung zur Effizienz

Die hier angeführten Optimierungsregeln produzieren sicherlich nicht den *bestmöglichen* Code; aus

```
sort xs = reduce merge [] (map (:[]) xs)
```

wird nach Optimierung durch `map`/`reduce`-deforestation

```
sort xs = h xs []
    where h [] acc = acc
          h (e:es) acc = h es (merge acc [e])
```

Während der erste Ausdruck MergeSort zu implementieren scheint (sortierte Teillisten werden verschmolzen), liegt der optimierten Version InsertionSort zugrunde. Dies zeigt, daß es ohne Annotationen im Programm nicht allgemein möglich ist, alle Ausdrücke zufriedenstellend zu optimieren; insbesondere können Optimierungen schlechte Algorithmen nicht in gute verwandeln (andererseits werden inhärent parallele Algorithmen auch nicht sequentialisiert).

5.8. Korrektheit

Der Compiler hat keine Möglichkeit, einen Operator auf seine Assoziativität zu überprüfen; aus diesem Grunde ist es möglich, daß die folgenden beiden Ausdrücke verschiedene Ergebnisse liefern, obwohl ihre Argumente semantisch identisch sind:

```
reduce (-) 0 [1..10]
reduce (-) 0 ([1..5]++[6..10])
```

Wegen der `map`/`reduce`-Optimierung ist die Reduktion sehr empfindlich, was die Art angeht, in welcher ihre Argumente berechnet werden (diese Empfindlichkeit verschwindet, wenn `reduce` korrekt angewendet wird, kann ansonsten jedoch zu schwer lokalisierbaren Fehlern führen).

5.9. Anwendungen des Mechanismus'

Der `map`/`reduce`-Mechanismus wird verwendet, um zusätzliche Spracheigenschaften von pH zu implementieren; hierbei wurde Wert darauf gelegt, jede der drei folgenden Eigenschaften, welche aus der Programmsprache Id stammen, in einheitlicher Weise zu behandeln, ohne ihre Effizienz zu mindern.

5.9.1. Schleifenkonstrukte

Der pH-Compiler enthält mehrere syntaktische Erweiterungen gegenüber Haskell; eine davon ist die `for`-Schleife, welche in Id als Folge von Bindungen ausgedrückt wird, welche freie Bezüge zu einer Menge von k zirkulärenden Variablen $c_1, \ldots, c_k$ und einer zugehörigen Menge von Nachfolge-Variablen $next_{c_1}, \ldots, next_{c_k}$ besitzt. Eine `for`-Schleife über die Liste `xs` läßt sich deshalb ausdrücken durch

```
let (anfängliche Bindungen für c1,...,ck)
in  for x <- xs do
      (verschiedene Bindungen, u.a. für next_c1,...,next_ck)
finally f c1,...,ck
```

Dieses Konstrukt läßt sich in eine Formulierung mit `foldr` überführen:

```
let (anfängliche Bindungen für c1,...,ck)
in  foldr (\ x cont c1,...,ck ->
      let (verschiedene Bindungen, u.a. für next_c1,...,next_ck)
      in  next_c1,...,next_ck
```

Durch Verwendung von inlining und η-Abstraktion (Hinzufügen) aller Argumente, welche zirkulierende Variablen repräsentieren, wird hierdurch vollständig iterativer Code erzeugt.

5.9.2. array comprehensions

Feld-Operationen sind in Haskell „schwerfälliger" als in Id, da einige von ihnen eine Liste von Index/Wert-Paaren als Argument erfordern. Die Erzeugung einer Multiplikationstabelle ließe sich formulieren als

```
table = array b [(i,a*b) | i@(a,b) <- range b] where b = ((0,0),(10,10))
```

pH orientiert sich aus diesem Grund an den von Id „geerbten" array comprehensions, welche I-Strukturen verwenden, um Felder parallel aufbauen zu können (durch Definition eines I-Struktur-verwendenden Listendurchlaufs); obwohl die konkrete Implementation (wie auch bei den im nächsten Abschnitt betrachteten offenen Listen) durch den Compiler verdeckt wird, wird sie durch die folgenden Definitionen charakterisiert:

```
emptyIArray :: Ix a => (a,a) -> IArray a b          -- empty IArray given bounds
writeIArray :: Ix a => IArray a b -> a -> b -> (b) -- write b at index a, then ()
readIArray  :: Ix a => IArray a b -> a -> b         -- read from index a
closeIArray :: Ix a => IArray a b -> Array a b      -- convert IArray into array
```

```
array bnds bindings = let iarr = emptyIArray bnds
                 sideEffect =
                        reduce (\ sideEffect1 sideEffect2 -> ()) ()
                               (map (\ (indx:=val) ->
                                         writeIArray iarr indx val)
                               (someOrder bindings))
                    in iarr
```


Hierbei ist zu beachten, daß die `reduce` übergebene Funktion *nichts* macht — ihre beiden Argumente müssen lediglich auf Seiteneffekte geprüft werden. Dies drückt aus, daß jede Zuweisung im Feld vollständig unabhängig ist von einer anderen (es sei denn, Indizes würden sich überschneiden, in welchem Fall das Programm jedoch sowieso fehlerhaft wäre); außerdem kann das Feld durch Verwendung eines I-Struktur-Modells und paralleler Erstellung und Verwendung des Felds zurückgeliefert werden, bevor der `reduce`-Ausdruck vollständig auf Seiteneffekte geprüft wurde (eine konkrete Transformation des obigen Beispiels findet sich in (Maessen, 1994)).

5.9.3. Offene Listen

Offene Listen werden intern durch Datenstrukturen dargestellt, welche isomorph zu (normalen) Listen sind; der Unterschied besteht darin, daß ihre Enden anfänglich leer sind und durch spätere I-Struktur-Operationen aufgefüllt werden können. Dies erlaubt sowohl die sequentielle Berechnung einzelner Listenelemente (von Anfang bis Ende) als auch die unabhängige (und damit potentiell parallele!) Berechnung von Teillisten, welche in konstanter Zeit miteinander verbunden werden können. Wie bei array comprehensions verdeckt der Compiler die konkrete Implementation hinter den folgenden Definitionen:

```
emptyOpen :: () -> Open a                 -- return empty open list
(>:)      :: Open a -> a -> Open a        -- add element to tail
(++:)     :: Open a -> Open a -> Open a   -- append tail to head
closeOpen :: Open a -> Open a -> [a]      -- convert given head and tail
```

Das Problem hierbei besteht darin, daß offene Listen keine wohldefinierte Identität besitzen; während sich ein *Aufruf* von `emptyOpen` als Identität verhält, wenn er als eines der beiden Argumente von `(++:)` erscheint, gibt es kein unterscheidbares *Objekt*, welches in dieser Weise korrekt verwendet werden kann.

Eine Teillösung besteht im Ausdruck der `map`/`reduce`-Optimierung durch Operationen auf offenen Listen durch

a_{list} = `(++:)`; u_{list} = $(\lambda e\ t.e$`>:`$t)$; t_{list} = `emptyOpen`

Da offene Listen einen vollständig anderen Typ besitzen als normale Listen, ist es notwendig, den Gebrauch von offenen Listen einzuschränken.

5.10. Einschränkungen

Die oben beschriebenen Optimierungen entfernen Listen, welche sofort verarbeitet werden, nachdem sie generiert wurden. Während dies eine nützliche Klasse von Listenoperationen einschließt und es erlaubt, Datenstrukturen (u.a. Felder und offene Listen) mittels list comprehensions auszudrücken, gibt es einige Anwendungsbeispiele welche hiermit *nicht* optimiert werden können:

- Neben den oben betrachteten Funktionen `map` und `concat` gibt es noch andere Funktionen, welche bei Eingabe einer Liste eine andere zurückzuliefern und nicht durch erstere ausgedrückt werden können (ein Beispiel hierfür ist die Funktion `init`, welche das letzte Listenelement entfernt)

- Mehrere Durchläufe des gleichen Arguments können prinzipiell kombiniert werden; wenn dies möglich ist, die Durchläufe sich jedoch unterscheiden (beispielsweise durch gleichzeitige Verwendung von `foldl` und `reduce`), so ist nicht klar, wie dies *effizient* geschehen kann. Dieses Problem tritt auf, wenn einer der zu kombinierenden Durchläufe Gebrauch macht von Funktionen höherer Ordnung (wie dies der Fall ist für `for`-Schleifen, array comprehensions oder offene Listen)

- Das simultane Durchlaufen zweier Listen ($\Longrightarrow$ `zip`) ist genauso problematisch; Maessen (1995) vermutet, dieses Problem könnte u.U. dadurch gelöst werden, daß die „Optimierungsrichtung" umgekehrt wird (ausgehend von der Generierung oder dem Gebrauch einer Liste, durch Kombination von `map` und `build`, bis die resultierende Liste durch `reduce`, `fold` oder `zip` verarbeitet wird) — dieser Ansatz wäre jedoch sehr schwer mit einer Lösung des Mehrfach-Durchlauf-Problems zu kombinieren

A. Beispiele

A.1. Entfernen einer rekursiven Datenstruktur

Wir betrachten im nachfolgenden exemplarisch den Ausdruck `sum (from n m)`, welcher im Gegensatz zum einführenden Beispiel `sumSq` nur eine zwischengeschaltete Liste erzeugt.

A.1.1. Vorgehensweise von (higher order) deforestation

```
sum_from n m = sum (from n m)
sum xs = case xs of
    [] -> 0
    (x:xs') -> x + sum xs'
from x y =
    if x > y
    then []
    else x : from (x+1) y
```

Wir können `sum` und `from` entfalten, um die Fallunterscheidung bzw. `(:)` und `[]` (Cons und Nil) zu enthüllen und erhalten:

```
sum_from n m = if n > m
    then case [] of
        [] -> 0
        (x:xs') -> x + sum xs'
    else case n : from (n+1) m of
        [] -> 0
        (x:xs') -> x + sum xs'
```

Es gibt zwei Instanzen der *Fallunterscheidungs-Reduktions-Transformation*, eine mit explizitem `[]` und eine mit explizitem `(:)`. Diese Transformation nimmt Ausdrücke der Form

$$\begin{array}{l} \texttt{case}\ \mathsf{Cons}\ e_1\ e_2\ \ldots\ e_n\ \texttt{of} \\ \quad\vdots \\ \quad \mathsf{Cons}\ v_1\ v_2\ \ldots\ v_n\ \texttt{->}\ e \\ \quad\vdots \end{array}$$

und transformiert diese in $\texttt{let}\ \{v_1 = e_1; v_2 = e_2; \ldots; v_n = e_n\}\ \texttt{in}\ e$, wodurch die (explizit zwischengeschalteten) Cons entfernt werden.

Eine Anwendung dieser Transformation, um diese zwischengeschalteten Konstruktoren zu entfernen, liefert:

```
sum_from n m =
    if n > m
    then 0
    else n + sum (from (n+1) m)
```


Wir haben den ersten Konstruktor in der zwischengeschalteten Liste zwischen `sum` und `from` eliminiert, aber eine neue Instanz von `sum (from ...)` ist aufgetaucht. Wiederholte Anwendung dieser Konstruktor-Entfernungstechnik würde weitere `(:)`-Zellen entfernen, *aber nur eine endliche Anzahl*. Außerdem würde das Programm mit jedem Schritt vergrößert werden.

Der Trick bei diesem Ansatz besteht darin, die Beziehung zwischen den verschiedenen Iteration dieser Listen-Entfernungstechnik zu erkennen und die Originaldefinition in die „aufgerollte" Version zurückzufalten. Dies kann eine effiziente, rekursive Definition ergeben, in welcher die zwischengeschaltete Liste entfernt wurde.

Im obigen Beispiel wissen wir von der Definition von `sum_from`, daß

$$\forall n\,.\,\forall m\,.\ \texttt{sum\_from}\ n\ m\ =\ \texttt{sum}\ \ (\texttt{from}\ n\ m)$$

Also können wir mit dieser Gleichung den Ausdruck `sum (from (n+1) m)` durch `sum_from (n+1) m` ersetzen; diese Ersetzung ergibt eine Definition von `sum_from`, welche ohne zwischengeschaltete Liste auskommt.

A.1.2. Vorgehensweise von cheap deforestation

Wir verwenden Funktionsdefinitionen in unserer kanonischen Form (siehe Abbildung 4), was eine gute Produktion und Konsumierung zwischengeschalteter Listen erlaubt:

```
sum xs = foldr (+) 0 xs
from x y =
    build (\ c n ->
        let
            from' x y =
                if x > y
                then n
                else x `c` from' (x+1) y
        in
            from' x y)
sum_from n m = sum (from n m)
```

Wir können nun `sum` und `from` in die rechte Seite von `sum_from` entfalten:

```
sum_from n m = foldr (+) 0 (build (\ c n ->
    let
        from' x y =
            if x > y
            then n
            else x `c` from' (x+1) y
    in
        from' n m))
```

Anwendung der `foldr`/`build`-Regel und β-Reduktion[6] liefert uns:

```
sum_from n m =
    let
        from' x y =
            if x > y
```

[6]Ersetzen eines Ausdrucks der Form $(\lambda v \rightarrow e_1)\ e_2$ durch den Ausdruck $\texttt{let}\ v\ =\ e_2\ \texttt{in}\ e_1$ und Einsetzen (*Inlining*) von v

```
        then 0
        else x + from' (x+1) y
in
    from' n m
```

A.2. Erzeugung eines parallelen Programms durch Optimierung

Wir betrachten die Anweisung `llsum = sum . concat`, welche alle Elemente einer Liste von Listen aufsummiert. Durch Einsetzen der Funktion `sum` und der Komposition erhalten wir die folgende Definition:

```
llsum xss = reduce (+) 0 (someOrder (concat xss))
```

Eine nützliche Optimierung beginnt, wenn die Reduktion bearbeitet wird (die Null als Konstante wird hier der Kürze wegen direkt eingesetzt):

$$\texttt{llsum xss = } \mathcal{LE}[\![\mathcal{LF}[\![\texttt{someOrder (concat xss)}]\!]\ (\lambda l\ r.(l\ 0)\ \texttt{+}\ (r\ 0))\ \texttt{const}\ 0]\!]$$

Optimierung fährt fort mit dem Aufruf von `someOrder`:

$$\texttt{llsum xss = } \mathcal{LE}[\![\mathcal{LF}[\![\texttt{concat xss}]\!]\ (\texttt{bothways}\ (\lambda l\ r\ t.(l\ 0)\ \texttt{+}\ (r\ 0)))\ \texttt{const}\ 0]\!]$$

Weiter mit `concat`:

$$\texttt{llsum xss = } \mathcal{LE}[\![\mathcal{LF}[\![\texttt{xss}]\!]\ (\lambda e.\mathcal{LF}[\![\texttt{e}]\!]\ (\texttt{bothways}\ \ldots)\ \texttt{const}\ 0)(\texttt{bothways}\ \ldots)\ \texttt{const}\ 0]\!]$$

Um nun die Schleife über `xss` zu generieren, muß die innere Optimierung eifrig (*eagerly*) ausgeführt werden, damit ihre Striktheit ermittelt werden kann. Um dies zu erreichen, verwenden wir die Hilfsvariable `d` für t; das Symbol `e` wird von der Übersetzung gestellt:

$$\mathcal{LF}[\![\texttt{e}]\!](\texttt{bothways}(\lambda l\ r\ t.(l\ 0)\ \texttt{+}\ (r\ 0)))\ \texttt{const}\ 0\ \texttt{d}$$

Da die Addition in beiden Argumenten strikt ist, kann der innere Durchlauf wie folgt iterativ durchgeführt werden:

```
let h [] acc = acc
    h (e:es) acc = h es (acc + e)
in h e d
```

Da `h` in `acc` strikt ist, war die ursprüngliche Funktion in ihrem letzten Argument strikt; der folgende Code für `llsum` ergibt deshalb:

```
llsum xss = LE⟦h' xss 0
    where h' [] acc' = acc'
          h' (x:xs) acc' = h' xs (acc' + h x 0)
              where h [] acc = acc
                    h (e:es) acc = h es (acc + e)⟧
```

Da nun keine weitere Optimierung im Funktionsrumpf möglich ist, verschwindet der Aufruf von $\mathcal{LE}$. Die wichtige Zeile in dieser Definition ist die zweite Regel von `h'`. In einem System, welches die Eigenschaften der Reduktion nicht kennt, würde für diese Zeile folgender Code erzeugt:

```
h' (x:xs) acc' = h' xs (h x acc')
```

Im `map`/`reduce`-System kann der Aufruf `h x 0` ausgeführt werden, bevor `acc'` verfügbar ist. Somit können durch eifriges Ausführen der äußeren Schleife `h'` die Teillisten unabhängig voneinander berechnet werden; nur für die Berechnung des letzten Schleifenergebnisses muß auf die Abarbeitung der inneren Schleifen gewartet werden.

A.3. Das Damenproblem

Dieses Programm wird vollständig durch Listen ausgedrückt, weswegen es großen Nutzen aus Optimierungen ziehen kann und gerne als Benchmark herangezogen wird (z.B. Gill, Launchbury und Peyton-Jones, 1993; Maessen, 1994; Marlow, 1995). Es summiert die Positionen der Damen in jeder Lösung auf und liefert diese Summe zurück; dies erzwingt die vollständige Auswertung jeder Damenposition in jeder Lösung, minimiert jedoch gleichzeitig die Kosten für eine Ausgabe des Ergebnisses:

```
main = (print.sum.concat.queens) 10

queens :: Int -> [[Int]]
queens k = queens_l k
    where
        queens_l :: Int -> [[Int]]
        queens_l 0 = [[]]
        queens_l m = [ q ++ [n] | q <- queens_l (m-1),
                                  n <- [1..k],
                                  safe q n]

        safe :: [Int] -> Int -> Bool
        safe q n = and [ (j /= n) && ( i+j /= m+n )
                                  && ( i-j /= m-n )
                                 | (i,j) <- zip [1..m-1] q ]
            where m = length q + 1
```

Die Ergebnisse der beschriebenen deforestation-Algorithmen sind dementsprechend „dramatisch" (siehe Tabelle 1; die dort angegebenen Zahlen sind natürlich nicht direkt vergleichbar). Wenn die Performance-Steigerungen in den meisten Anwendungen auch wesentlich geringer ausfallen werden, so zeigt dieses Beispiel jedoch eindrucksvoll, welches Potential diese Algorithmen besitzen (tatsächlich gibt es „Real-World"-Anwendungen, welche *noch* eindrucksvoller sind; Gill (1996) führt hier beispielsweise einen funktionalen Raytracer an)

Algorithmus aus	Compiler	Vor Optimierung		Nach Optimierung	
		Speicherbedarf	Ausführungszeit	Speicherbedarf	Ausführungszeit
(Marlow, 1995)	GHC 0.23	134.0 MB	16.51 s	19.4 MB	6.16 s
(Gill et al., 1993)	GHC 0.23	179.0 MB	24.40 s	36.0 MB	8.80 s
(Gill, 1996)	GHC 0.26	61.3 MB	9.40 s	15.6 MB	5.40 s
(Maessen, 1994)	hbc	190.0 MB	45.40 s	66.0 MB	25.90 s

Tabelle 1: Speicherbedarf/Ausführungszeiten bei Anwendung der deforestations

B. Begriffserläuterungen

higher-order function (Funktion höherer Ordnung) Eine FhO. ist eine Funktion, welche ihrerseits eine Funktion als Argument besitzt oder als Ergebnis liefert; sie wird automatisch durch das „currying" von Argumenten definiert. Ein Vorteil der FhO. ist, daß sie die Formulierung algebraischer Eigenschaften und deren Verwendung bei Manipulationen von Ausdrücken erleichtert.

Transparenz T. ist die Eigenschaft einer Optimierungstechnik, welche nicht allgemein anwendbar ist, daß der Programmierer genau weiß, wann sein Programm optimiert werden wird.

Je mächtiger die Optimierungen sind, umso wichtiger ist T.: Nicht-transparente Optimierungen tendieren dazu, unvorhersehbare Ergebnisse zu erzeugen; eine kleine Änderung im ursprünglichen Programm kann große Unterschide in der Performance erzeugen, da die Anwendbarkeit der Optimierung geändert wird.

Eine Daumenregel-Definition von T., welche subjektiven Charakter besitzt, ist folgende: Wenn eine Optimierung garantiert, anwendbar zu sein bei Programmen, welche ein syntaktisches Kriterium erfüllen, dann ist sie transparent.

intermediate data structures (zwischengeschaltete Datenstrukturen) Strukturen, welche für interne Berechnungen verwendet werden, aber nicht Bestandteil des Endergebnisses sind

source-to-source-Transformation Bei einer stsT. wird das Programm in einer dem Programmierer genehmen Form geschrieben und danach in eine für die Maschine genehme Form transformiert; diese Technik läßt sich für jede Programmiersprache anwenden, besonders gut jedoch für funktionelle Sprachen, welche durch einfache Argumentation über Gleichungen manipuliert werden können (Gleiches wird durch Gleiches ersetzt)

C. Das Unfold/Fold-Transformationssystem

Das Unfold/Fold-Transformationssystem von Burstall/Darlington war die Basis für die meisten Programm-Transformationstechniken, welche zur Optimierung funktionaler Programme entwickelt wurden.

Das System besteht aus sechs grundlegenden Regeln, welche auf eine Menge von Gleichungen angewendet werden: Die ursprüngliche Gleichungsmenge ist hierbei das zu optimierende Programm; weitere Gleichungen können jedoch während der Transformation hinzukommen.

Jede Gleichung besitzt eine linke Seite der Form $v\,(e_1, \ldots, e_n)$, wobei die rechte Seite ein beliebiger Ausdruck sein kann (dies ist ähnlich den pattern-matching-Funktionsdefinitionen, wie sie in vielen funktionalen Sprachen vorkommen).

Die Regeln selbst, welche auf das Programm angewendet werden können, lauten wie folgt:

1. *Definition.* Eine neue Gleichung mit einer linken Seite, welche sich noch nicht in der aktuellen Menge befindet, wird eingefügt.

2. *Instanziierung (Instantiation).* Eine neue Gleichung, welche eine Substitutions-Instanz einer existierenden Gleichung darstellt, wird eingefügt.

3. *Entfaltung (Unfolding).* Wenn $e = e'$ und $f = f'$ Gleichungen sind und f' eine Substitutions-Instanz von e, Se, enthält, kann sie durch Se' ersetzt werden.

4. *Faltung (Folding).* Wenn $e = e'$ und $f = f'$ Gleichungen sind und f' eine Substitutions-Instanz von e', Se', enthält, kann sie durch Se ersetzt werden.

5. *Abstraktion (Abstraction).* Eine `where`-Klausel kann in jede Gleichung $e = e'$ eingefügt werden durch Ersetzung der rechten Seite mit $e'[x_1/f_1 \ldots x_n/f_n]$ `where` $(x_1, \ldots, x_n) = (f_1, \ldots, f_n)$

6. *Gesetze (Laws).* Alle Gesetze über Primitive, welche in Ausdrücken verwendet werden können (z.B. Assoziativität, Kommutativität, usw.)

Diese sechs Gesetze sind ausreichend allgemein, um jedes Programm in fast jedes andere Programm zu transformieren. Das System ist nicht völlig vollständig, da es Funktionen wie $f([\,]) = 0, f(x : xs) = f(xs)$ nicht in $f(xs) = 0$ transformieren kann; es gibt jedoch eine zusätzliche Regel, genannt *Redefinition*, welche solche Transformationen erlaubt.

Das o.g. Transformationssystem ist nur teilweise korrekt, da es möglich ist, daß ein aus dem ursprünglichen Programm durch Transformationen erhaltenes Programm schwächer definiert ist als das Original. Dies kann nur geschehen, wenn ein beliebiger Gebrauch des fold-Schritts erlaubt ist: So kann z.B. eine fold-Regel eine Gleichung $f(n) = e$ in der Menge durch $f(n) = f(n)$ ersetzen — offensichtlich eine nichtterminierende Funktion.

(Die Korrektheit ist hingegen garantiert, wenn die Anzahl der unfold-Schritte immer größer ist als die Anzahl der fold-Schritte, sofern bestimmte Bedingungen erfüllt sind.)

Literaturverzeichnis

Bird, R. und Wadler, P. (1988). *Introduction to Functional Programming*, Prentice Hall.

Gill, A. J. (1996). *Cheap Deforestation for Non-strict Functional Languages*, PhD thesis, University of Glasgow Department of Computing Science.

Gill, A. J., Launchbury, J. und Peyton-Jones, S. L. (1993). A Short Cut To Deforestation, *in* Arvind (Hrsg.), *Functional Programming and Computer Architecture*, ACM SIGPLAN/SIGARCH, ACM, Kopenhagen, Dänemark, S. 223–232.

Gill, A. J. und Peyton-Jones, S. L. (1994). Cheap Deforestation in Practice: An Optimiser for Haskell, *Proc. (IFIP)*, Vol. 1, Hamburg, S. 581–586.

Hudak, P. und Fasel, J. H. (1992). A Gentle Introduction to Haskell, *ACM SIGPLAN Notices*, Vol. 27, ACM.
URL: *http://haskell.cs.yale.edu/cgi-alingelb/aviewcgi?/~alingelb/tutorial.html*

Maessen, J.-W. (1994). *Eliminating intermediate lists in pH using local transformations*, Master's thesis, MIT Department of Electrical Engineering and Computer Science.

Maessen, J.-W. (1995). Simplifying Parallel List Traversal, *Technical Report CSG Memo 370*, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

Marlow, S. D. (1995). *Deforestation for Higher-Order Functional Programs*, PhD thesis, University of Glasgow Department of Computing Science.

Marlow, S. D. und Wadler, P. (1996). Higher Order Deforestation.
URL: *ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Simon_Marlow/paper.ps.gz*

Wadler, P. (1990). Deforestation: Transforming programs to eliminate trees, *Theoretical Computer Science* **73**: 231–248.