Type-Inference Based Deforestation of Functional Programs

Olaf Chitil

Lehrstuhl für Informatik II, RWTH Aachen

University of York

Modularität durch Datenorientierte Programmierung

```
factorial :: Int → Int
factorial n = prod [1..n]

any :: (α → Bool) → [α] → Bool
any p xs = or (map p xs)

anyTree :: (α → Bool) → Tree α → Bool
anyTree p = any p . treeToList

intToDigits :: Int → [Int]
intToDigits = map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)
```

Strikte vs. Nicht-Strikte Auswertung

```
   any (> 2) [1,2,3,4,5,6,7,8,9]
⇝
   or (map (> 2) [1,2,3,4,5,6,7,8,9])
⇝
   or [False, False, True, True, True, True, True, True, True]
⇝
   True
```

```
   any (> 2) [1,2,3,4,5,6,7,8,9]
⇝
   or (map (> 2) [1,2,3,4,5,6,7,8,9])
⇝
   or (False : (map (> 2) [2,3,4,5,6,7,8,9]))
⇝
   or (map (> 2) [2,3,4,5,6,7,8,9])
⇝
   or (False : (map (> 2) [3,4,5,6,7,8,9]))
⇝
   or (map (> 2) [3,4,5,6,7,8,9])
⇝
   or (True : (map (> 2) [4,5,6,7,8,9]))
⇝
   True
```

Deforestation

```
any p xs = or (map p xs)
```

```
any p []     = False
any p (x:xs) = p x || any p xs
```

Eliminiert Zwischendatenstrukturen durch Fusion von Erzeuger und Verbraucher.

Short Cut Deforestation [Gill, Launchbury, Peyton Jones 93]

Nur Verbrauch durch `foldr`

- einheitliches Ersetzen der Listenkonstruktoren

$$[x_1, x_2, x_3, \ldots, x_k]$$

$$\equiv$$

$$x_1 \texttt{:} (x_2 \texttt{:} (x_3 \texttt{:} \ldots (x_k \texttt{:} \texttt{[]}) \ldots)))$$

$$\rightsquigarrow \texttt{foldr c n}$$

$$x_1 \texttt{ `c` } (x_2 \texttt{ `c` } (x_3 \texttt{ `c` } (\ldots (x_k \texttt{ `c` n}) \ldots)))$$

Short Cut Deforestation (Fortsetzung)

Ersetze nicht einfach alle Listenkonstruktoren im Erzeuger:

```
foldr (||) False (map even [1,2])
```

```
map even (1 || 2 || False)
```

Short Cut Deforestation (Fortsetzung)

Finde und markiere die Listenkonstruktoren, die die Zwischenliste aufbauen:

```
let map f []     = []
    map f (x:xs) = (f x) : (map f xs)
in map even (1 : 2 : [])
```

Abstrahiere von diesen Zwischenlistenkonstruktoren:

```
(λc.λn. let map f []     = n
            map f (x:xs) = (f x) `c` (map f xs)
        in map even (1 : 2 : []) )
  (:) []
```

Short Cut Fusionsregel

Short Cut Fusion:

```
foldr (||) False ((λc.λn. let map ...) (:) [])
```

```
(λc.λn. let map ...) (||) False
```

Allgemein:

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ (e_{\texttt{p}}\ \texttt{(:)}\ \texttt{[]}) \quad \rightsquigarrow \quad e_{\texttt{p}}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

wobei die Funktion $e_{\texttt{p}}$ von den Listenkonstruktoren abstrahiert, die die Zwischenliste erzeugen.

Ursprüngliche Short Cut Deforestation

Fusionsregel:

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ \texttt{(build}\ e_{\texttt{p)}} \quad \rightsquigarrow \quad e_{\texttt{p}}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

```
build g = g (:) []
```

In den Standardbibliotheken:

- alle listenproduzierenden Funktionen sind mit `build` definiert
- alle listenverbrauchenden Funktionen sind mit `foldr` definiert

```
map f xs = build (λc.λn. foldr (c . f) n xs)
```

Korrektheit von Short Cut Fusion

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ (e_{\texttt{p}}\ \texttt{(:)}\ \texttt{[]}) \quad \rightsquigarrow \quad e_{\texttt{p}}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

Nur, wenn $e_{\texttt{p}} :: \forall\gamma.\ (\tau \to \gamma \to \gamma) \to \gamma \to \gamma$

Ursprüngliche Short Cut Deforestation:

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ (\texttt{build}\ e_{\texttt{p}}) \quad \rightsquigarrow \quad e_{\texttt{p}}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

```
build :: (∀γ. (α → γ → γ) → γ → γ) → [α]
build g = g (:) []
```

Listenabstraktion durch Typinferenz

$\{$`inc` $::$ `Int` $\to$ `Int`, `1` $::$ `Int`, `2` $::$ `Int`,
`(:)` $::$ `Int` $\to$ `[Int]` $\to$ `[Int]`, `[]` $::$ `[Int]`$\}$

$\vdash$
```
let mapInt :: (Int → Int) → [Int] → [Int]
      mapInt f []     = []
      mapInt f (x:xs) = (f x) : (mapInt f xs)
  in mapInt inc (1 : (2 : []))
```
$::$ `[Int]`

Ersetze jeden Listenkonstruktor durch eine neue Variable:

$(:) \rightsquigarrow \mathtt{c}_i \qquad [] \rightsquigarrow \mathtt{n}_i$

$$
\begin{array}{l}
\{\mathtt{inc} :: \mathtt{Int} \to \mathtt{Int},\ 1 :: \mathtt{Int},\ 2 :: \mathtt{Int}, \\
\ \mathtt{c}_1 :: \mathtt{Int} \to [\mathtt{Int}] \to [\mathtt{Int}],\ \mathtt{c}_2 :: \mathtt{Int} \to [\mathtt{Int}] \to [\mathtt{Int}], \\
\ \mathtt{c}_3 :: \mathtt{Int} \to [\mathtt{Int}] \to [\mathtt{Int}],\ \mathtt{n}_1 :: [\mathtt{Int}],\ \mathtt{n}_2 :: [\mathtt{Int}]\} \\
\vdash \mathtt{let\ mapInt} :: (\mathtt{Int} \to \mathtt{Int}) \to [\mathtt{Int}] \to [\mathtt{Int}] \\
\qquad\quad \mathtt{mapInt\ f\ []} \quad\ = \mathtt{n}_1 \\
\qquad\quad \mathtt{mapInt\ f\ (x:xs)} = \mathtt{(f\ x)}\ `\mathtt{c}_1`\ \mathtt{(mapInt\ f\ xs)} \\
\quad \mathtt{in\ mapInt\ inc\ (1}\ `\mathtt{c}_2`\ \mathtt{(2}\ `\mathtt{c}_3`\ \mathtt{n}_2\mathtt{))}
\end{array}
$$

Ersetze jeden Listentyp durch eine neue Variable:

$\texttt{[Int]} \rightsquigarrow \gamma_i$

$\{\texttt{inc} :: \texttt{Int} \to \texttt{Int},\ \texttt{1} :: \texttt{Int},\ \texttt{2} :: \texttt{Int},$
$\texttt{c}_1 :: \texttt{Int} \to \gamma_4 \to \gamma_4,\ \texttt{c}_2 :: \texttt{Int} \to \gamma_5 \to \gamma_5,\ \texttt{c}_3 :: \texttt{Int} \to \gamma_6 \to \gamma_6,$
$\texttt{n}_1 :: \gamma_7,\ \texttt{n}_2 :: \gamma_8\}$
$\vdash$ `let mapInt :: (Int → Int) → γ1 → γ2`
`mapInt f [] = n1`
`mapInt f (x:xs) = (f x) ‘c1‘ (mapInt f xs)`
`in mapInt inc (1 ‘c2‘ (2 ‘c3‘ n2))`

Typinferenzalgorithmus

ersetzt einige Typvariablen, um (allgemeinste) gültige Typung zu erhalten

$\{\texttt{inc} :: \texttt{Int} \to \texttt{Int},\ 1 :: \texttt{Int},\ 2 :: \texttt{Int},$
$\texttt{c}_1 :: \texttt{Int} \to \gamma \to \gamma,\ \texttt{c}_2 :: \texttt{Int} \to \texttt{[Int]} \to \texttt{[Int]},\ \texttt{c}_3 :: \texttt{Int} \to \texttt{[Int]} \to \texttt{[Int]},$
$\texttt{n}_1 :: \gamma,\ \texttt{n}_2 :: \texttt{[Int]}\}$
$\vdash$

```
let mapInt :: (Int → Int) → [Int] → γ
      mapInt f []     = n1
      mapInt f (x:xs) = (f x) `c1` (mapInt f xs)
  in mapInt inc (1 `c2` (2 `c3` n2))
```

$:: \gamma$

Bilde listenabstrahierten Erzeuger:

$\{\texttt{inc} :: \texttt{Int} \to \texttt{Int},\ \texttt{1} :: \texttt{Int},\ \texttt{2} :: \texttt{Int}\ \}$

$\vdash \lambda \texttt{c} : \texttt{Int} \to \gamma \to \gamma.\ \lambda \texttt{n} : \gamma.$

```
    let mapInt :: (Int → Int) → [Int] → γ
        mapInt f []     = n
        mapInt f (x:xs) = (f x) `c` (mapInt f xs)
    in mapInt inc (1 : (2 : []))
```

$:: \forall\gamma.\ (\texttt{Int} \to \gamma \to \gamma) \to \gamma \to \gamma$

Inlinen

```
mapInt inc (1 : (2 : []))
```

Typinferenz stellt fest, daß `mapInt` inlined werden muß:

$$\{\texttt{inc} :: \texttt{Int} \to \texttt{Int},\ 1 :: \texttt{Int},\ 2 :: \texttt{Int},$$
$$\texttt{mapInt} :: (\texttt{Int} \to \texttt{Int}) \to \gamma_1 \to \gamma_2,$$
$$\texttt{c}_1 :: \texttt{Int} \to \gamma_1 \to \gamma_1,\ \texttt{c}_2 :: \texttt{Int} \to \gamma_1 \to \gamma_1,\ \texttt{n}_1 :: \gamma_1\}$$
$$\vdash \texttt{mapInt inc (1 `c}_1\texttt{` (2 `c}_2\texttt{` n}_1\texttt{))}$$
$$:: \gamma_2$$

Probleme:

- umfangreiches Inlinen über Modulgrenzen nicht praktikabel
- Inlinen erfordert delikate Kompromisse um Codeexplosion zu vermeiden

Aufteilung in Worker und Wrapper

```
mapInt :: (Int → Int) → [Int] → [Int]
mapInt f []     = []
mapInt f (x:xs) = (f x) : (mapInt f xs)
```

Worker-Funktion, die von der Ergebnisliste abstrahiert:

```
mapIntW :: ∀γ. (Int → γ → γ)→ γ →(Int → Int) → [Int] → γ
mapIntW c n f []     = n
mapIntW c n f (x:xs) = (f x) `c` (mapIntW c n f xs)
```

Wrapper-Funktion:

```
mapInt = mapIntW (:) []
```

Wrapper reichen für Short Cut Fusion

Wrapper sind klein genug, um überall inlined zu werden.

```
mapInt inc (1 : (2 : []))
```

Inlinen des Wrappers

```
mapIntW (:) [] inc (1 : (2 : []))
```

Listenabstraktion ohne Inlinen

$$\lambda \texttt{c} : \texttt{Int} \to \gamma \to \gamma . \lambda \texttt{n} : \gamma . \texttt{mapIntW c n inc (1 : (2 : []))}$$

Mehr Worker und Wrapper

```
appW :: ∀α.∀γ.(α → γ → γ)→ γ →[α]→ γ → γ
appW c n xs ys = foldr c ys xs

(++) :: ∀α.[α]→[α]→[α]
(++) = appW (:) []

unzipW :: ∀α.∀β.∀γ.∀δ.(α→γ→γ)→ γ →(β→δ→δ)→ δ →[(α,β)]→(γ,δ)
unzipW = ...

unzip :: ∀α.∀β.[(α,β)]→([α],[β])
unzip = unzipW (:) [] (:) []
```

Funktionen, die ihr eigenes Ergebnis verbrauchen

```
reverse :: ∀α.[α] → [α]
reverse []     = []
reverse (x:xs) = (++) (reverse xs) (x : [])
```

↝ Aufteilung in Worker/Wrapper

```
reverseW :: ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
reverseW c n []     = n
reverseW c n (x:xs) = appW c n (reverseW (:) [] xs) (c x n)
reverse = reverseW (:) []
```

↝ Inlinen von `appW` und Short Cut Fusion

```
...
reverseW c n (x:xs) = reverseW c (c x n) xs
```

Sprache und Typinferenzalgorithmus

Sprache: Erweiterter getypter λ-Kalkül zweiter Ordnung.

- kleine Sprache
- explizite und einfache Handhabung von Typen
- Haskell in diese Sprache übersetzbar
- Variante der im Glasgow Haskell Compiler verwendeten Zwischensprache

Typinferenzalgorithmus: Variante des Hindley-Milner Algorithmus $\mathcal{W}$.

- $\mathcal{W} : (\Gamma, e) \mapsto (\tau, \sigma)$ so daß $\Gamma\sigma \vdash e\sigma :: \tau$ gültig
- funktioniert hier, da Eingabe wohlgetypt und mit Typannotationen versehen
- Erweiterung nötig, um bei geschachtelten Listen geeignete Typen zu inferieren

Praxis

- Separater Prototyp implementiert
- Von Hand transformiert und mit Glasgow Haskell Compiler gemessen: 10-Damen-Problem

Version	Heap alloziert (MB)	Zeit (s)
ursprünglich, gem. Haskell Report	205.78	1.71
ursprünglich, realistisch	98.48	1.01
Short Cut Deforestation des GHC	33.41	0.57
Typinferenz-basierte Deforestation	22.43	0.51

- Implementierung im GHC auf Basis des Prototypen in Arbeit
  Kirsten Chevalier, Franklyn Turbak (Wellesley, Massachusetts)

Zusammenfassung

Automatische Listenabstraktion durch Typinferenz
anstelle anzugebender, restriktiverer `build`-Form.

Algorithmus:

1. ▷ teile alle listenerzeugenden Funktionen in Worker und Wrapper auf
   ▷ inline alle Wrapper
2. ▷ suche Anwendungen von `foldr`
   ▷ abstrahiere Zwischenliste vom Erzeuger
   ▷ fusioniere Verbraucher mit Erzeuger