Deforestation von Funktionalen Programmen durch Typinferenz

Olaf Chitil

RWTH Aachen

Modularität durch Datenorientierte Programmierung

```
factorial :: Int -> Int
factorial n = prod [1..n]

any :: (a -> Bool) -> [a] -> Bool
any p xs = or (map p xs)

anyTree :: (a -> Bool) -> Tree a -> Bool
anyTree p = any p . treeToList

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

Deforestation

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

$$\rightsquigarrow$$

```
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

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

$$\equiv$$

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

$$\Big\downarrow \mathtt{foldr\ c\ n}$$

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

Short Cut Deforestation (Fortsetzung)

Ersetze nicht einfach alle Listenkonstruktoren im Erzeuger:

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

$\not\rightsquigarrow$

```
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:

```
(\c n -> 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 ((\c n -> let map ...) (:) [])
```

$$\rightsquigarrow$$

```
(\c n -> 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 (\c n -> 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 :: (forall c. (a->c->c) -> c -> c) -> [a]
build g = g (:) []
```

Listenabstraktion durch Typinferenz

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

```
⊢ 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:

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

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

Ersetze jeden Listentyp durch eine neue Variable:

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

$$
\begin{aligned}
&\{\texttt{inc} :: \texttt{Int} \to \texttt{Int},\ \texttt{1} :: \texttt{Int},\ \texttt{2} :: \texttt{Int},\\
&\ \texttt{c}_1 :: \texttt{Int} \to \gamma_4 \to \gamma_4,\ \texttt{c}_2 :: \texttt{Int} \to \gamma_5 \to \gamma_5,\ \texttt{c}_3 :: \texttt{Int} \to \gamma_6 \to \gamma_6,\\
&\ \texttt{n}_1 :: \gamma_7,\ \texttt{n}_2 :: \gamma_8\}\\
&\vdash \texttt{let mapInt :: (Int} \to \texttt{Int)} \to \gamma_1 \to \gamma_2\\
&\qquad\quad \texttt{mapInt f []} \qquad\ \ \texttt{=}\ \texttt{n}_1\\
&\qquad\quad \texttt{mapInt f (x:xs) = (f x) `c}_1\texttt{` (mapInt f xs)}\\
&\quad\ \texttt{in mapInt inc (1 `c}_2\texttt{` (2 `c}_3\texttt{` n}_2\texttt{))}
\end{aligned}
$$

Typinferenzalgorithmus

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

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

$\ \texttt{c}_1 :: \texttt{Int} \to \gamma \to \gamma,\ \texttt{c}_2 :: \texttt{Int} \to \texttt{[Int]} \to \texttt{[Int]},\ \texttt{c}_3 :: \texttt{Int} \to \texttt{[Int]} \to \texttt{[Int]},$

$\ \texttt{n}_1 :: \gamma,\ \texttt{n}_2 :: \texttt{[Int]}\}$

$\vdash$ `let mapInt :: (Int` $\to$ `Int)` $\to$ `[Int]` $\to$ 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:

$\{\mathtt{inc} :: \mathtt{Int} \rightarrow \mathtt{Int},\ \mathtt{1} :: \mathtt{Int},\ \mathtt{2} :: \mathtt{Int}\ \}$

$\vdash \lambda\mathtt{c} : \mathtt{Int} \rightarrow \gamma \rightarrow \gamma.\ \lambda\mathtt{n} : \gamma.$

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

$:: \forall\gamma.\ (\mathtt{Int} \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma$

Inlinen

$$\texttt{mapInt inc (1 : (2 : []))}$$

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

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

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)
```

$\rightsquigarrow$

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 : []))`

$\rightsquigarrow$ Inlinen des Wrappers

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

$\rightsquigarrow$ Listenabstraktion ohne Inlinen

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

Mehr Worker und Wrapper

$\texttt{appW :: } \forall\alpha.\forall\gamma.(\alpha \to \gamma \to \gamma) \to \gamma \to [\alpha] \to \gamma \to \gamma$

```
appW c n xs ys = foldr c ys xs
```

$\texttt{(++) :: } \forall\alpha.[\alpha] \to [\alpha] \to [\alpha]$

```
(++) = appW (:) []
```

$\texttt{unzipW :: } \forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha \to \gamma \to \gamma) \to \gamma \to (\beta \to \delta \to \delta) \to \delta \to [(\alpha,\beta)] \to (\gamma,\delta)$

```
unzipW = ...
```

$\texttt{unzip :: } \forall\alpha.\forall\beta.[(\alpha,\beta)] \to ([\alpha],[\beta])$

```
unzip = unzipW (:) [] (:) []
```

Funktionen, die ihr eigenes Ergebnis verbrauchen

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

$\rightsquigarrow$ 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 (:) []
```

$\rightsquigarrow$ 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

Problem bei geschachtelten Listen

$$\{\texttt{xs} :: \texttt{[[Int]]}\} \vdash \texttt{[] : xs} :: \texttt{[[Int]]}$$

$\leadsto$ Konstruktoren Ersetzen

$$\{\texttt{xs} :: \texttt{[[Int]]},\ \texttt{c} :: \texttt{[Int]} \rightarrow \texttt{[[Int]]} \rightarrow \texttt{[[Int]]},\ \texttt{n} :: \texttt{[[Int]]}\} \vdash \texttt{c n xs}$$

$\leadsto$ Typen Ersetzen

$$\{\texttt{xs} :: \texttt{[[Int]]},\ \texttt{c} :: \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2,\ \texttt{n} :: \gamma_3\} \vdash \texttt{c n xs}$$

$\leadsto$ Typinferenz

$$\{\texttt{xs} :: \texttt{[[Int]]},\ \texttt{c} :: \gamma_1 \rightarrow \texttt{[[Int]]} \rightarrow \texttt{[[Int]]},\ \texttt{n} :: \gamma_1\} \vdash \texttt{c n xs} :: \texttt{[[Int]]}$$

$\leadsto$?

Ursache: $\texttt{(:)} :: \forall \alpha . \alpha \rightarrow [\alpha] \rightarrow [\alpha]$

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

Zukünftige Arbeiten

- Implementierung
- Worker/Wrapper-Schema für Verbraucher
- Verallgemeinerung auf andere Datenstrukturen
- effzientere Aufteilung in Worker/Wrapper durch Integration von Typinferenz und Aufteilungsalgorithmus