Type Inference Builds a Short Cut to Deforestation

Olaf Chitil

RWTH Aachen

Deforestation

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

$\rightsquigarrow$

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

Eliminate intermediate data structure by fusion of producer with consumer.

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

Only consumation by `foldr`

- uniform replacement of list constructors

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

$$\equiv$$

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

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

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

Short Cut Deforestation (continued)

Don't just replace all list constructors in the producer:

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

$$\not\rightsquigarrow$$

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

Short Cut Deforestation (continued)

Find and mark the list constructors that build the intermediate list:

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

Abstract from intermediate list constructors:

```
(\c n -> let map f []     = n
             map f (x:xs) = (f x) `c` (map f xs)
           in map even (1 : 2 : []) )
  (:) []
```

Short cut fusion:

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

```
(\c n -> let map ...) (||) False
```

Short Cut Fusion Rule

Idea:

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ \texttt{((\textbackslash c n -> }e'\texttt{) (:) [])} \quad \rightsquigarrow \quad \texttt{(\textbackslash c n -> }e'\texttt{)}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

Correct rule in $\lambda 2$:

for all $e_{\texttt{(:)}} : \tau_1 \to \tau_2 \to \tau_2,\quad e_{\texttt{[]}} : \tau_2,\quad e_{\texttt{p}} : \forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma$

$$\boxed{\begin{array}{c}\texttt{foldr}\ \tau_1\ \tau_2\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ (e_{\texttt{p}}\ [\tau_1]\ \texttt{((:)}\ \tau_1\texttt{)}\ \texttt{([]}\ \tau_1\texttt{))} \\ \rightsquigarrow \\ e_{\texttt{p}}\ \tau_2\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\end{array}}$$

here always: $e_{\texttt{p}} = \lambda\gamma.\lambda\texttt{c} : \tau_1 \to \gamma \to \gamma.\lambda\texttt{n} : \gamma.\ e'$

Original Short Cut Deforestation

In the standard libraries:

- all list consuming functions defined in terms of `foldr`
- all list producing functions defined in terms of `build`

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

Fusion rule:

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

Example:

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

List Abstraction through Type Inference

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

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

Replace list constructors and list types by new variables:

$$\texttt{(:)} \rightsquigarrow \texttt{c}_i \quad (\texttt{Int} \to \gamma_i \to \gamma_i), \qquad \texttt{[]} \rightsquigarrow \texttt{n}_i \quad (\gamma_i), \qquad \texttt{[Int]} \rightsquigarrow \gamma_i$$

$$\begin{array}{l}
\{\texttt{inc} : \texttt{Int} \to \texttt{Int},\ \texttt{1} : \texttt{Int},\ \texttt{2} : \texttt{Int}, \\
\ \texttt{c}_1 : \texttt{Int} \to \gamma_4 \to \gamma_4,\ \texttt{c}_2 : \texttt{Int} \to \gamma_5 \to \gamma_5,\ \texttt{c}_3 : \texttt{Int} \to \gamma_6 \to \gamma_6,\ \texttt{n}_1 : \gamma_7,\ \texttt{n}_2 : \gamma_8\} \\
\vdash \texttt{let mapInt} : (\texttt{Int} \to \texttt{Int}) \to \gamma_1 \to \gamma_2 \\
\qquad\quad \texttt{mapInt f []} \qquad\quad \texttt{=}\ \texttt{n}_1 \\
\qquad\quad \texttt{mapInt f (x:xs) = (f x) `}\texttt{c}_1\texttt{` (mapInt f xs)} \\
\qquad \texttt{in mapInt inc (1 `}\texttt{c}_2\texttt{` (2 `}\texttt{c}_3\texttt{` }\texttt{n}_2\texttt{))} \\
: \gamma_3
\end{array}$$

Type inference algorithm

instantiates type variables to obtain a (most general) valid typing

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

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

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

$\vdash$ `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))
```

$: \gamma$

Obtain list abstracted producer:

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

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

Inlining

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

Result of type inference:

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

So `mapInt` needs to be inlined.

A wrapper/worker scheme avoids inlining of large expressions [IFL'99].

List Abstraction Example

$$\{\texttt{fst}:\forall\alpha.\forall\beta.(\alpha\texttt{,}\beta) \to \alpha,\ \texttt{zs}:\texttt{[(Bool, Char)]},$$
$$\texttt{unzip}:\forall\alpha.\forall\beta.\texttt{[(}\alpha\texttt{,}\beta\texttt{)]} \to \texttt{([}\alpha\texttt{],[}\beta\texttt{])}\}$$
$$\vdash \texttt{fst [Bool] [Char] (unzip Bool Char zs)}$$
$$\texttt{: [Bool]}$$

$$\rightsquigarrow$$

$$\lambda\beta.\ \lambda\texttt{c}:\texttt{Bool}\to\beta\to\beta.\ \lambda\texttt{n}:\beta.$$

```
let unzip' : [(Bool,Char)]→(β,[Char])
           = foldr (Bool,Char) (β,[Char])
               (λy:(Bool,Char).λu:(β,[Char]).
                 ...
in fst β [Char] (unzip' zs)
```

Type Inference Algorithm

$$\Gamma \vdash e : \tau \quad \rightsquigarrow \quad \sigma \text{ with } \Gamma\sigma \vdash e\sigma : \tau\sigma \text{ valid}$$

- variation of Hindley-Milner type inference algorithm
  - ▷ based on $\mathcal{M} : (\Gamma, e, \tau) \mapsto \sigma$ [Lee,Yi 1998]
    instead of $\mathcal{W} : (\Gamma, e) \mapsto (\tau, \sigma)$
- can work on $\lambda 2$, because
  - ▷ type abstractions and type applications are given
  - ▷ $\Gamma\hat{\sigma} \vdash e\hat{\sigma} : \tau\hat{\sigma}$ is valid for $\hat{\sigma}$
    with $\alpha\hat{\sigma} = [\rho]$ for all $\alpha \in \mathrm{freeTyVar}(\Gamma, e)$
    (i.e., all universal type quantifications in type annotations are given)

Conclusions

- New Deforestation Algorithm
  - search for application of foldr
  - abstract intermediate list from producer by using type inference
  - fuse consumer with producer
- simple and efficient
- type-inference based abstraction applicable to other data structures than lists
- to do: implementation in the Glasgow Haskell compiler