Type-Inference Based Short Cut Deforestation (nearly) without Inlining

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

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

$$\equiv$$

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

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

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

Short Cut Deforestation (continued)

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

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

$$\not\leadsto$$

```
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 ...) (:)  [])
```

$$\rightsquigarrow$$

```
(\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 $e_{\texttt{(:)}} : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2, \quad e_{\texttt{[]}} : \tau_2, \quad e_{\texttt{p}} : \forall\gamma.(\tau_1 \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma$

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

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

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 special function `build`

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

List Abstraction through Type Inference

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

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

$$\{\texttt{inc} : \texttt{Int} \to \texttt{Int},\ \texttt{1} : \texttt{Int},\ \texttt{2} : \texttt{Int},$$
$$\texttt{c}_1 : \texttt{Int} \to \gamma_4 \to \gamma_4,\ \texttt{c}_2 : \texttt{Int} \to \gamma_5 \to \gamma_5,\ \texttt{c}_3 : \texttt{Int} \to \gamma_6 \to \gamma_6,\ \texttt{n}_1 : \gamma_7,\ \texttt{n}_2 : \gamma_8\}$$

$\vdash$ `let mapInt :` $(\texttt{Int} \to \texttt{Int}) \to \gamma_1 \to \gamma_2$

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

$: \gamma_3$

Type inference algorithm

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

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

$\vdash$ `let mapInt :` $(\texttt{Int} \to \texttt{Int}) \to \texttt{[Int]} \to \gamma$

```
      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:

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

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

Inlining

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

Type inference determines that `mapInt` needs to be inlined:

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

Problems:

- extensive inlining across module boundaries not practically feasible
- inlining requires delicate compromises to avoid code bloat

The Worker/Wrapper Split

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

$\rightsquigarrow$

worker function that abstracts from result list:

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

wrapper function:

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

Wrappers Suffice for Short Cut Fusion

Wrappers are small enough to be inlined everywhere.

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

$$\rightsquigarrow \text{ inlining of wrapper}$$

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

$$\rightsquigarrow \text{ list abstraction without inlining}$$

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

More Workers and Wrappers

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

$\texttt{appW } \alpha\ \gamma\ \texttt{c n xs ys = foldr } \alpha\ \gamma\ \texttt{c ys xs}$

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

$\texttt{(++) } \alpha\ \texttt{= appW } \alpha\ [\alpha]\ \texttt{((:) } \alpha\texttt{) ([] } \alpha\texttt{)}$

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

$\texttt{unzipW = } \ldots$

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

$\texttt{unzip } \alpha\ \beta\ \texttt{= unzipW } \alpha\ \beta\ [\alpha]\ [\beta]\ \texttt{((:) } \alpha\texttt{) ([] } \alpha\texttt{) ((:) } \beta\texttt{) ([] } \beta\texttt{)}$

Functions that Consume their own Result

$\texttt{reverse} : \forall\alpha.[\alpha] \to [\alpha]$

$\texttt{reverse}\ \alpha\ \texttt{[]} \qquad = \texttt{[]}\ \alpha$

$\texttt{reverse}\ \alpha\ \texttt{(x:xs)} = \texttt{(++)}\ \alpha\ \texttt{(reverse}\ \alpha\ \texttt{xs)}\ \texttt{((:)}\ \alpha\ \texttt{x}\ \texttt{([]}\ \alpha\texttt{))}$

$\rightsquigarrow$ worker/wrapper split

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

$\texttt{reverseW}\ \alpha\ \gamma\ \texttt{c n []} \qquad = \texttt{n}$

$\texttt{reverseW}\ \alpha\ \gamma\ \texttt{c n (x:xs)} = \texttt{appW}\ \alpha\ \gamma\ \texttt{c n (reverseW}\ \alpha\ [\alpha]\ \texttt{((:)}\ \alpha\texttt{)}\ \texttt{([]}\ \alpha\texttt{) xs)}$
$\qquad\qquad\qquad\qquad\qquad\qquad \texttt{(c x n)}$

$\texttt{reverse}\ \alpha = \texttt{reverseW}\ \alpha\ [\alpha]\ \texttt{((:)}\ \alpha\texttt{)}\ \texttt{([]}\ \alpha\texttt{)}$

$\rightsquigarrow$ inlining of `appW` and short cut fusion

...

$\texttt{reverseW}\ \alpha\ \gamma\ \texttt{c n (x:xs)} = \texttt{reverseW}\ \alpha\ \gamma\ \texttt{c (c x n) xs}$

Type-Inference Based Short Cut Deforestation (nearly) without Inlining

1. ▷ split all list producing functions into workers and wrappers

   ▷ inline all wrappers

2. ▷ search for application of foldr

   ▷ abstract intermediate list from producer (without inlining)

   ▷ fuse consumer with producer

Future Work

- implement worker/wrapper split
  - as prototype
  - in the Glasgow Haskell compiler
- more efficient worker/wrapper split through integration of splitting and type inference
- worker/wrapper scheme for consumers
- generalisation to other data types than lists