ICFP, October 4, 2002

Concatenate, Reverse and Map Vanish For Free

Janis Voigtländer

Dresden University of Technology

`http://wwwtcs.inf.tu-dresden.de/~voigt`

Supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-1 and by the "Gesellschaft von Freunden und Förderern der TU Dresden" with a travel grant.

List-Producers using $+\!\!+$, *reverse* and *map*:

$$part\ even\ [1..10] = [2,4,6,8,10,1,3,5,7,9]$$

$$
\begin{array}{l}
part :: (\alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\
part\ p\ l = \text{let}\ f\ []\ z = z \\
\qquad\qquad\quad f\ (x : xs)\ z = \text{if}\ p\ x\ \text{then}\ x : (f\ xs\ z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else}\ f\ xs\ (z +\!\!+ [x]) \\
\qquad\qquad \text{in}\ f\ l\ []
\end{array}
$$

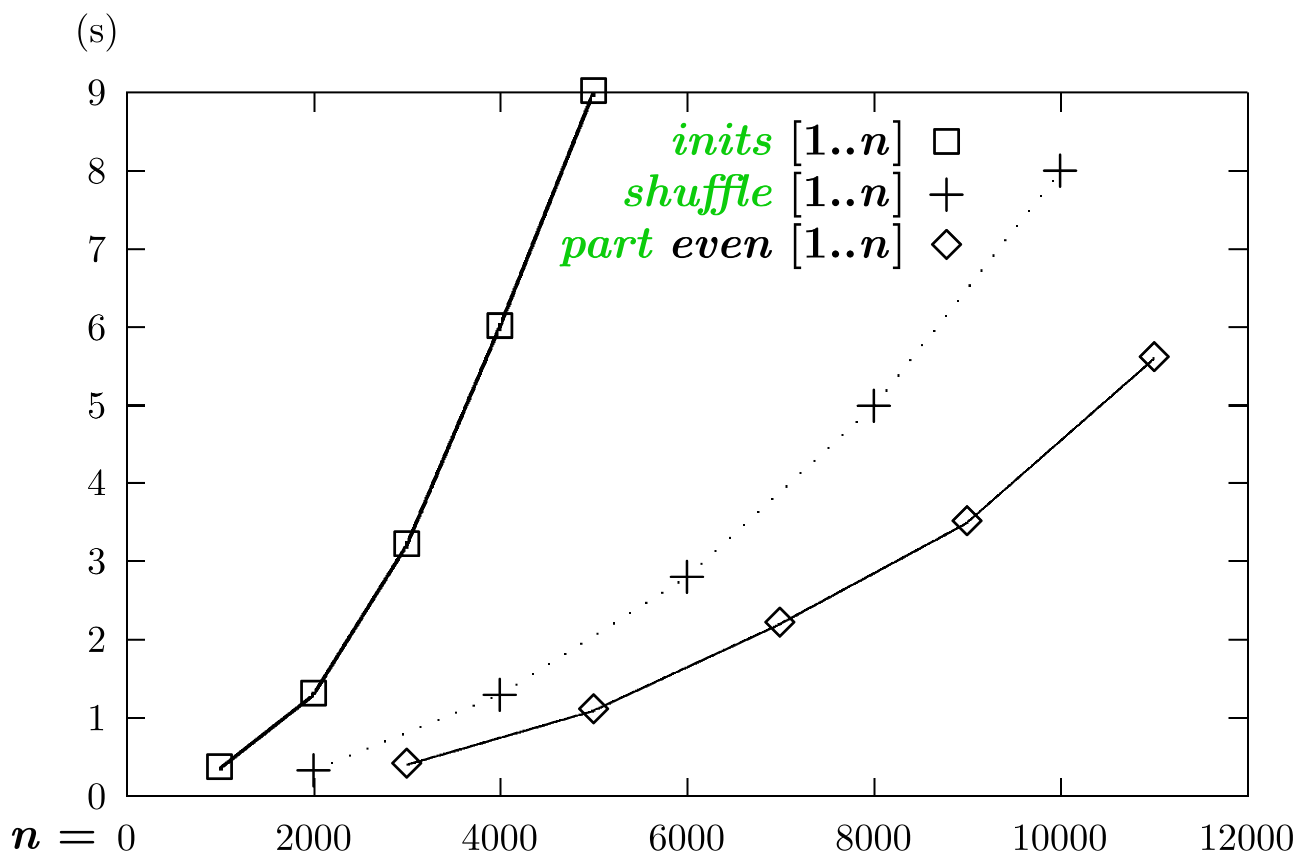
$$shuffle\ \text{“whatever”} = \text{“waeervth”}$$

$$
\begin{array}{l}
shuffle :: [\alpha] \rightarrow [\alpha] \\
shuffle\ [] = [] \\
shuffle\ (x : xs) = x : (reverse\ (shuffle\ xs))
\end{array}
$$

$$inits\ [1..4] = [[],[1],[1,2],[1,2,3],[1,2,3,4]]$$

$$
\begin{array}{l}
inits :: [\alpha] \rightarrow [[\alpha]] \\
inits\ [] = [[]] \\
inits\ (x : xs) = [] : (map\ (x :)\ (inits\ xs))
\end{array}
$$

Runtimes dominated by repeated List-Operations:



Efficiency by List Abstraction (*part*):

$$
\begin{aligned}
&\mathit{part} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\mathit{part}\ p\ l = \text{let } f\ [\,]\ z = z \\
&\qquad\qquad\qquad f\ (x : xs)\ z = \text{if } p\ x \text{ then } x : (f\ xs\ z) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } f\ xs\ (z +\!\!+ (x : [\,])) \\
&\qquad\qquad\quad \text{in } f\ l\ [\,]
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
&\mathit{part}^{\star} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\mathit{part}^{\star}\ p\ l = \mathit{vanish}_{+\!+}\ (\lambda n\ c\ a \to \\
&\qquad\qquad\quad \text{let } f\ [\,]\ z = z \\
&\qquad\qquad\qquad f\ (x : xs)\ z = \text{if } p\ x \text{ then } x\ \text{`}c\text{`}\ (f\ xs\ z) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } f\ xs\ (z\ \text{`}a\text{`}\ (x\ \text{`}c\text{`}\ n)) \\
&\qquad\qquad\quad \text{in } f\ l\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{vanish}_{+\!+} :: (\forall \beta\,.\,\beta \to (\alpha \to \beta \to \beta) \\
&\qquad\qquad\qquad \to (\beta \to \beta \to \beta) \to \beta) \to [\alpha] \\
&\mathit{vanish}_{+\!+}\ g = g\ [\,]\ (:)\ (+\!\!+)
\end{aligned}
$$

Such list abstraction can be performed *automatically*, based on the rank-2 polymorphic type of $\mathit{vanish}_{+\!+}$ and partial type inference [Chitil, 1999]!

Runtimes: $n =$	3000	5000	7000	9000	11000
part *even* $[1..n]$	0.4	1.1	2.2	3.5	5.6 (s)
$\mathit{part}^{\star}$ *even* $[1..n]$	0.004	0.006	0.009	0.012	0.015 (s)

Efficiency by List Abstraction (*shuffle*):

$$
\begin{aligned}
&\mathit{shuffle} :: [\alpha] \longrightarrow [\alpha] \\
&\mathit{shuffle}\ []\ = []\\
&\mathit{shuffle}\ (x : xs) = x : (\mathit{reverse}\ (\mathit{shuffle}\ xs))
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
&\mathit{shuffle}^\star :: [\alpha] \longrightarrow [\alpha] \\
&\mathit{shuffle}^\star\ l = \mathit{vanish}_{rev}\ (\lambda n\ c\ r \longrightarrow \\
&\qquad \text{let } f\ []\ = n \\
&\qquad\quad f\ (x : xs) = x\ `c`\ (r\ (f\ xs)) \\
&\qquad \text{in } f\ l)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{vanish}_{rev} :: (\forall \beta\,.\,\beta \longrightarrow (\alpha \longrightarrow \beta \longrightarrow \beta) \longrightarrow (\beta \longrightarrow \beta) \longrightarrow \beta) \longrightarrow [\alpha] \\
&\mathit{vanish}_{rev}\ g \sqsupseteq g\ []\ (:)\ \mathit{reverse}
\end{aligned}
$$

Runtimes: $n =$	2000	4000	6000	8000	10000
shuffle $[1..n]$	0.33	1.3	2.8	5.0	8.0 (s)
shuffle* $[1..n]$	0.005	0.01	0.016	0.02	0.025 (s)

Efficiency by List Abstraction (*inits*):

$$\begin{aligned}
&inits :: [\alpha] \to [[\alpha]]\\
&inits\ [] = [] : []\\
&inits\ (x : xs) = [] : (map\ (x :)\ (inits\ xs))
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
&inits^{\star} :: [\alpha] \to [[\alpha]]\\
&inits^{\star}\ l = vanish_{++,rev,map}\ (\lambda n\ c\ a\ r\ m \to\\
&\qquad \text{let } f\ [] = []\ `c`\ n\\
&\qquad\qquad f\ (x : xs) = []\ `c`\ (m\ (x :)\ (f\ xs))\\
&\qquad \text{in } f\ l)
\end{aligned}$$

$$\begin{aligned}
&vanish_{++,rev,map} :: (\forall\beta\ .\ \cdots) \to [\alpha]\\
&vanish_{++,rev,map}\ g \sqsupseteq g\ []\ (:)\ (++)\ reverse\ map
\end{aligned}$$

Runtimes: $n =$	1000	2000	3000	4000	5000
inits $[1..n]$	0.35	1.3	3.2	6.0	9.0 (s)
inits* $[1..n]$	0.08	0.3	0.7	1.3	2.0 (s)

Actual Definitions of the *vanish*-Combinators:

$$vanish_{++} :: (\forall \beta . \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \beta) \to [\alpha]$$
$$vanish_{++}\ g \ = \ g\ id\ (\lambda x\ h\ ys \to x : (h\ ys))\ (\circ)\ []$$

$$vanish_{rev} :: (\forall \beta . \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta) \to \beta) \to [\alpha]$$
$$\begin{aligned} vanish_{rev}\ g \ = \ & fst\ (g\ (\lambda ys \to (ys, ys)) \\ & \quad (\lambda x\ h\ ys \to (x : (fst\ (h\ ys)), snd\ (h\ (x : ys)))) \\ & \quad (\lambda h\ ys \to swap\ (h\ ys))\ []) \end{aligned}$$

$$\begin{aligned} vanish_{++,rev,map} :: \ & (\forall \beta . \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to (\beta \to \beta) \\ & \to ((\alpha \to \alpha) \to \beta \to \beta) \to \beta) \to [\alpha] \end{aligned}$$
$$\begin{aligned} vanish_{++,rev,map}\ g \ = \ & fst\ (g\ (\lambda f\ ys \to (ys, ys)) \\ & \quad (\lambda x\ h\ f\ ys \to ((f\ x) : (fst\ (h\ f\ ys)), \\ & \qquad\qquad snd\ (h\ f\ ((f\ x) : ys)))) \\ & \quad (\lambda h_1\ h_2\ f\ ys \to (fst\ (h_1\ f\ (fst\ (h_2\ f\ ys))), \\ & \qquad\qquad snd\ (h_2\ f\ (snd\ (h_1\ f\ ys))))) \\ & \quad (\lambda h\ f\ ys \to swap\ (h\ f\ ys)) \\ & \quad (\lambda k\ h\ f\ ys \to h\ (f \circ k)\ ys)\ id\ []) \end{aligned}$$

User-Exposed Semantics of the *vanish*-Combinators:

$$vanish_{++} :: (\forall \beta \,.\, \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta) \rightarrow [\alpha]$$
$$vanish_{++}\ g \ = \ g\ []\ (:)\ (++)$$

$$vanish_{rev} :: (\forall \beta \,.\, \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta) \rightarrow [\alpha]$$
$$vanish_{rev}\ g \ \sqsupseteq \ g\ []\ (:)\ reverse$$

$$vanish_{++,rev,map} :: (\forall \beta \,.\, \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta) \rightarrow \beta) \rightarrow [\alpha]$$
$$vanish_{++,rev,map}\ g \ \sqsupseteq \ g\ []\ (:)\ (++)\ reverse\ map$$

Proven using *free theorems* [Wadler, 1989], driven by the algebraic laws:

$$(xs \mathbin{++} ys) \mathbin{++} zs = xs \mathbin{++} (ys \mathbin{++} zs) \qquad (1)$$
$$reverse\ (reverse\ xs) \sqsubseteq xs \qquad (2)$$
$$map\ f\ (map\ k\ xs) = map\ (f \circ k)\ xs \qquad (3)$$

Proof: $vanish_{++}\ g = g\ []\ (:)\ (++)$

Parametricity [Reynolds, 1983] gives for the type of

$$g :: \forall \beta\ .\ \beta \rightarrow (\mathsf{A} \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta$$

the following *free theorem* [Wadler, 1989]:

$$\begin{aligned} & (n, n') \in \mathcal{R} \ \wedge\ (\forall x :: \mathsf{A}, (l, l') \in \mathcal{R}\ .\ (c\ x\ l, c'\ x\ l') \in \mathcal{R}) \\ \wedge\ & (\forall (l_1, l_1') \in \mathcal{R}, (l_2, l_2') \in \mathcal{R}\ .\ (a\ l_1\ l_2, a'\ l_1'\ l_2') \in \mathcal{R}) \\ \Rightarrow\ & (g\ n\ c\ a, g\ n'\ c'\ a') \in \mathcal{R}. \end{aligned}$$

Instantiate with $n = []$, $c = (:)$, $a = (++)$, $n' = id$, $c' = (\lambda x\ h\ ys \rightarrow x : (h\ ys))$, $a' = (\circ)$, and $\mathcal{R} = \{(l, l') \mid \forall ys :: [\mathsf{A}]\ .\ l ++ ys = l'\ ys\}$:

$$\begin{aligned} & (\forall ys\ .\ [] ++ ys = ys) \\ \wedge\ & (\forall x, l, l'\ .\ (\forall ys\ .\ l ++ ys = l'\ ys) \ \Rightarrow\ (\forall ys\ .\ (x : l) ++ ys = x : (l'\ ys))) \\ \wedge\ & (\forall l_1, l_1', l_2, l_2'\ .\ (\forall ys\ .\ l_1 ++ ys = l_1'\ ys) \ \wedge\ (\forall ys\ .\ l_2 ++ ys = l_2'\ ys) \\ & \quad \Rightarrow\ (\forall ys\ .\ (l_1 ++ l_2) ++ ys = l_1'\ (l_2'\ ys))) \\ \Rightarrow\ & (\forall ys\ .\ (g\ []\ (:)\ (++)) ++ ys = g\ id\ (\lambda x\ h\ ys \rightarrow x : (h\ ys))\ (\circ)\ ys). \end{aligned}$$

The preconditions of this implication are fulfilled by the definition of $(++)$ and by law (1), hence: $(g\ []\ (:)\ (++)) ++ [] = vanish_{++}\ g$.

A general Methodology (e.g.: the *filter* vanishes)

$$\begin{aligned}
&nub :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \\
&nub\ [] = [] \\
&nub\ (x : xs) = x : (filter\ (x \neq)\ (nub\ xs))
\end{aligned}$$

1. Freezing and Efficient Conversion:

$$\begin{aligned}
&\text{data } \mathsf{List}\ \alpha = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\mathsf{List}\ \alpha) \mid \mathsf{Filter}\ (\alpha \rightarrow \mathsf{Bool})\ (\mathsf{List}\ \alpha) \\
&nub' :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow \mathsf{List}\ \alpha \\
&nub'\ [] = \mathsf{Nil} \\
&nub'\ (x : xs) = \mathsf{Cons}\ x\ (\mathsf{Filter}\ (x \neq)\ (nub'\ xs))
\end{aligned}$$

$$\begin{aligned}
&convert^\star :: \mathsf{List}\ \alpha \rightarrow [\alpha] \\
&convert^\star\ l = \text{let } h\ \mathsf{Nil}\ p = [] \\
&\qquad h\ (\mathsf{Cons}\ x\ xs)\ p = \text{if } (p\ x) \text{ then } (x : (h\ xs\ p)) \text{ else } (h\ xs\ p) \\
&\qquad h\ (\mathsf{Filter}\ q\ xs)\ p = h\ xs\ (\lambda x \rightarrow q\ x\ \&\&\ p\ x) \\
&\quad \text{in } h\ l\ (\lambda x \rightarrow \mathsf{True})
\end{aligned}$$

2. Preparing Shortcut Fusion [Gill *et al.*, 1993]:

$$build_{\mathsf{List}}\ g \ = \ g\ \mathsf{Nil}\ \mathsf{Cons}\ \mathsf{Filter}$$

$$\begin{aligned}
&nub' :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to \mathsf{List}\ \alpha\\
&nub'\ l = build_{\mathsf{List}}\ (\lambda n\ c\ f \to \text{let } h\ [\,] = n\\
&\qquad\qquad\qquad\qquad\qquad\quad h\ (x : xs) = c\ x\ (f\ (x \neq)\ (h\ xs))\\
&\qquad\qquad\qquad\qquad\quad \text{in } h\ l)
\end{aligned}$$

$$\begin{aligned}
fold_{\mathsf{List}}\ \mathsf{Nil}\ n\ c\ f &= n\\
fold_{\mathsf{List}}\ (\mathsf{Cons}\ x\ xs)\ n\ c\ f &= c\ x\ (fold_{\mathsf{List}}\ xs\ n\ c\ f)\\
fold_{\mathsf{List}}\ (\mathsf{Filter}\ q\ xs)\ n\ c\ f &= f\ q\ (fold_{\mathsf{List}}\ xs\ n\ c\ f)
\end{aligned}$$

$$\begin{aligned}
&convert^{\star} :: \mathsf{List}\ \alpha \to [\alpha]\\
&convert^{\star}\ l = fold_{\mathsf{List}}\ l\\
&\qquad\qquad (\lambda p \to [\,])\\
&\qquad\qquad (\lambda x\ h\ p \to \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p))\\
&\qquad\qquad (\lambda q\ h\ p \to h\ (\lambda x \to q\ x\ \&\&\ p\ x))\\
&\qquad\qquad (\lambda x \to \mathsf{True})
\end{aligned}$$

3. Calculate using Fusion Law: $fold_{\mathsf{List}}\ (build_{\mathsf{List}}\ g) = g$

$$
\begin{aligned}
& convert^{\star}\ (nub'\ l) \\
= {} & fold_{\mathsf{List}}\ (build_{\mathsf{List}}\ (\lambda n\ c\ f \rightarrow \text{let } h\ [\,] = n \\
& \qquad\qquad\qquad\qquad\qquad\quad h\ (x : xs) = c\ x\ (f\ (x \neq)\ (h\ xs)) \\
& \qquad\qquad\qquad\qquad\qquad \text{in } h\ l)) \\
& \qquad (\lambda p \rightarrow [\,]) \\
& \qquad (\lambda x\ h\ p \rightarrow \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p)) \\
& \qquad (\lambda q\ h\ p \rightarrow h\ (\lambda x \rightarrow q\ x\ \&\&\ p\ x)) \\
& \qquad (\lambda x \rightarrow \mathsf{True}) \\
= {} & (\lambda n\ c\ f \rightarrow \text{let } h\ [\,] = n \\
& \qquad\qquad\quad h\ (x : xs) = c\ x\ (f\ (x \neq)\ (h\ xs)) \\
& \qquad\qquad \text{in } h\ l) \\
& (\lambda p \rightarrow [\,]) \\
& (\lambda x\ h\ p \rightarrow \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p)) \\
& (\lambda q\ h\ p \rightarrow h\ (\lambda x \rightarrow q\ x\ \&\&\ p\ x)) \\
& (\lambda x \rightarrow \mathsf{True})
\end{aligned}
$$

4. Abstract into Combinator:

$$vanish_{\mathit{filter}}\ g \;=\; g\ (\lambda p \rightarrow [\,])\ (\lambda x\ h\ p \rightarrow \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p))$$
$$(\lambda q\ h\ p \rightarrow h\ (\lambda x \rightarrow q\ x\ \&\&\ p\ x))\ (\lambda x \rightarrow \mathsf{True})$$

$$nub^{\star} :: \mathsf{Eq}\ \alpha \;\Rightarrow\; [\alpha] \;\rightarrow\; [\alpha]$$
$$\begin{aligned} nub^{\star}\ l \;=\; vanish_{\mathit{filter}}\ (\lambda n\ c\ f \rightarrow \text{let } & h\ [\,] = n \\ & h\ (x : xs) = c\ x\ (f\ (x \neq)\ (h\ xs)) \\ \text{in } & h\ l) \end{aligned}$$

5. Prove Correctness:

$$vanish_{\mathit{filter}} :: (\forall \beta . \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow ((\alpha \rightarrow \mathsf{Bool}) \rightarrow \beta \rightarrow \beta) \rightarrow \beta) \rightarrow [\alpha]$$
$$vanish_{\mathit{filter}}\ g \;=\; g\ [\,]\ (:)\ \mathit{filter}$$

Using a *free theorem* and the following law:

$$\mathit{filter}\ p\ (\mathit{filter}\ q\ xs) = \mathit{filter}\ (\lambda x \rightarrow q\ x\ \&\&\ p\ x)\ xs \qquad (4)$$

Proof: $\mathit{vanish}_{\mathit{filter}}\ g = g\ []\ (:)\ \mathit{filter}$

From the type of g follows the *free theorem*:

$$
\begin{array}{ll}
 & (n, n') \in \mathcal{R} \\
\wedge & (\forall x :: \mathsf{A}, (l, l') \in \mathcal{R}\ .\ (c\ x\ l, c'\ x\ l') \in \mathcal{R}) \\
\wedge & (\forall q :: \mathsf{A} \to \mathsf{Bool}, (l, l') \in \mathcal{R}\ .\ (f\ q\ l, f'\ q\ l') \in \mathcal{R}) \\
\Rightarrow & (g\ n\ c\ f, g\ n'\ c'\ f') \in \mathcal{R}.
\end{array}
$$

Instantiate with $n = []$, $c = (:)$, $f = \mathit{filter}$, $n' = (\lambda p \to [])$, $c' = (\lambda x\ h\ p \to \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p))$, $f' = (\lambda q\ h\ p \to h\ (\lambda x \to q\ x\ \&\&\ p\ x))$, and $\mathcal{R} = \{(l, l') \mid \forall p :: \mathsf{A} \to \mathsf{Bool}\ .\ \mathit{filter}\ p\ l = l'\ p\}$:

$$
\begin{array}{ll}
 & (\forall p\ .\ \mathit{filter}\ p\ [] = []) \\
\wedge & (\forall x, l, l'\ .\ (\forall p\ .\ \mathit{filter}\ p\ l = l'\ p) \\
 & \quad \Rightarrow (\forall p\ .\ \mathit{filter}\ p\ (x : l) = \text{if } (p\ x) \text{ then } (x : (l'\ p)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } (l'\ p))) \\
\wedge & (\forall q, l, l'\ .\ (\forall p\ .\ \mathit{filter}\ p\ l = l'\ p) \\
 & \quad \Rightarrow (\forall p\ .\ \mathit{filter}\ p\ (\mathit{filter}\ q\ l) = l'\ (\lambda x \to q\ x\ \&\&\ p\ x))) \\
\Rightarrow & (\forall p\ .\ \mathit{filter}\ p\ (g\ []\ (:)\ \mathit{filter}) \\
 & \qquad = g\ (\lambda p \to []) \\
 & \qquad\qquad (\lambda x\ h\ p \to \text{if } (p\ x) \text{ then } (x : (h\ p)) \text{ else } (h\ p)) \\
 & \qquad\qquad (\lambda q\ h\ p \to h\ (\lambda x \to q\ x\ \&\&\ p\ x))\ p).
\end{array}
$$

The preconditions of this implication are fulfilled by the definition of *filter* and by law (4), hence:

$$\mathit{filter}\ (\lambda x \to \mathsf{True})\ (g\ []\ (:)\ \mathit{filter}) = \mathit{vanish}_{\mathit{filter}}\ g.$$

Summary:

- **Variation of list abstraction: abstract not only over data constructors, but also over manipulating operations.**
- **Methodology: "freezing" plus "efficient conversion as a *fold*" for synthesizing optimized list implementations.**
  (also applicable to other algebraic data types)
- **Encapsulate essence of optimizations in reusable rank-2 polymorphic combinators.**

$\Rightarrow$ **Allows automation and proofs using free theorems.**

References

[Chitil, 1999] Chitil, O. (1999). Type inference builds a short cut to deforestation. *Pages 249–260 of: International Conference on Functional Programming, Paris, France.* ACM Press.

[Gill *et al.*, 1993] Gill, A., Launchbury, J., & Peyton Jones, S.L. (1993). A short cut to deforestation. *Pages 223–232 of: Functional Programming Languages and Computer Architecture, Copenhagen, Denmark.* ACM Press.

[Hughes, 1986] Hughes, R.J.M. (1986). A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, **22**, 141–144.

[Reynolds, 1983] Reynolds, J.C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: Information Processing, Paris, France.* Elsevier Science Publishers B.V.

[Wadler, 1987] Wadler, P. (1987). *The concatenate vanishes.* Note, University of Glasgow, Scotland.

[Wadler, 1989] Wadler, P. (1989). Theorems for free! *Pages 347–359 of: Functional Programming Languages and Computer Architecture, London, England.* ACM Press.