

# Translucent Abstraction

## Algebraic Datatypes with Safe Views

Meng Wang, Jeremy Gibbons

University of Oxford (Visiting NII)

# Algebraic Datatypes is Transparent and Open

**data** *List a = Nil | Cons a (List a)*

Pattern Matching:

*append :: (List a, List a) → List a*

*append (Nil, ys) = ys*

*append (Cons x xs, ys) = Cons x (append (xs, ys))*

Equational Reasoning:

*append (Cons 1 Nil, Cons 2 Nil)*

*= Cons 1 (append (Nil, Cons 2 Nil))*

*= Cons 1 (Cons 2 Nil)*

# Algebraic Datatypes is Transparent and Open

**data** *List a = Nil | Cons a (List a)*

Pattern Matching:

*append* :: (*List a, List a*) → *List a*

*append (Nil, ys) = ys*

*append (Cons x xs, ys) = Cons x (append (xs, ys))*

Equational Reasoning:

*append (Cons 1 Nil, Cons 2 Nil)*  
= *Cons 1 (append (Nil, Cons 2 Nil))*  
= *Cons 1 (Cons 2 Nil)*

# Algebraic Datatypes is Transparent and Open

**data** *List a = Nil | Cons a (List a)*

Pattern Matching:

*append* :: (*List a, List a*) → *List a*

*append (Nil, ys) = ys*

*append (Cons x xs, ys) = Cons x (append (xs, ys))*

Equational Reasoning:

$$\begin{aligned} & \textit{append} (\textit{Cons} \ 1 \ \textit{Nil}, \textit{Cons} \ 2 \ \textit{Nil}) \\ &= \textit{Cons} \ 1 \ (\textit{append} \ (\textit{Nil}, \textit{Cons} \ 2 \ \textit{Nil})) \\ &= \textit{Cons} \ 1 \ (\textit{Cons} \ 2 \ \textit{Nil}) \end{aligned}$$

# Algebraic Datatypes is Transparent and Open

**type** *Queue a = List a*

*emptyQ = Nil*

*enQ :: a → Queue a → Queue a*

*enQ x xs = append xs (Cons x Nil)*

*deQ :: Queue a → Queue a*

*deQ (Cons x xs) = xs*

# Algebraic Datatypes is Transparent and Open

**type** *Queue a = (List a, List a)*

*emptyQ = Nil*

*enQ :: a → Queue a → Queue a*

*enQ x xs = append xs (Cons x Nil)*

*deQ :: Queue a → Queue a*

*deQ (Cons x xs) = xs*

# Translucent Views

- Firstly introduced by Wadler (POPL 1987) and is an on-going research topic
- Translucency = Encapsulation + Openness

**data** *List a = Nil | Cons a (List a)*

**view** *List a = Lin | Snoc (List a) a*

*to Nil* = *Lin*

*to (Cons x Nil)* = *Snoc Nil x*

*to (Cons x (Snoc xs y))* = *Snoc (Cons x xs) y*

*from Lin* = *Nil*

*from (Snoc Nil x)* = *Cons x Nil*

*from (Snoc (Cons x xs) y)* = *(Cons x (Snoc xs y))*

# Translucent Views

- Firstly introduced by Wadler (POPL 1987) and is an on-going research topic
- Translucency = Encapsulation + Openness

**data**  $List\ a = Nil \mid Cons\ a\ (List\ a)$

**view**  $List\ a = Lin \mid Snoc\ (List\ a)\ a$

$to\ Nil = Lin$

$to\ (Cons\ x\ Nil) = Snoc\ Nil\ x$

$to\ (Cons\ x\ (Snoc\ xs\ y)) = Snoc\ (Cons\ x\ xs)\ y$

$from\ Lin = Nil$

$from\ (Snoc\ Nil\ x) = Cons\ x\ Nil$

$from\ (Snoc\ (Cons\ x\ xs)\ y) = (Cons\ x\ (Snoc\ xs\ y))$

# Translucent Views

**data** *List a = Nil | Cons a (List a)*

**view** *List a = Lin | Snoc (List a) a*

*to Nil* = *Lin*

*to (Cons x Nil)* = *Snoc Nil x*

*to (Cons x (Snoc xs y))* = *Snoc (Cons x xs) y*

*from Lin* = *Nil*

*from (Snoc Nil x)* = *Cons x Nil*

*from (Snoc (Cons x xs) y)* = *(Cons x (Snoc xs y))*

*init (Snoc xs x) = xs*

*reverse Nil* = *Nil*

*reverse (Cons x xs) = Snoc (reverse xs) x*

# Equational Reasoning

$$\begin{aligned} \text{to Nil} &= \text{Lin} \\ \text{to (Cons x Nil)} &= \text{Snoc Nil x} \\ \text{to (Cons x (Snoc xs y))} &= \text{Snoc (Cons x xs) y} \\ \text{from Lin} &= \text{Nil} \\ \text{from (Snoc Nil x)} &= \text{Cons x Nil} \\ \text{from (Snoc (Cons x xs) y)} &= (\text{Cons x (Snoc xs y)}) \\ \text{reverse Nil} &= \text{Nil} \\ \text{reverse (Cons x xs)} &= \text{Snoc (reverse xs) x} \end{aligned}$$

$$\begin{aligned} &\text{reverse (Cons 1 (Cons 2 Nil))} = \text{Snoc (reverse (Cons 2 Nil)) 1} \\ &= \text{Snoc (Snoc Nil 2) 1} = \text{Snoc (Cons 2 Nil) 1} \\ &= \text{Cons 1 (Snoc Nil 2)} = \text{Cons 2 (Cons 1 Nil)} \end{aligned}$$

# Translucent Views

- Firstly introduced by Wadler (POPL 1987) and is an on-going research topic
- Translucency = Encapsulation + Openness

**type** *Queue a = (List a, List a)*

**view** *Queue a = List a*

*to = append*

*from = ?*

# Bidirectional Transformation

A function maps source into view:

$$\textit{get} :: S \rightarrow V$$

The backward transformation:

$$\textit{put} :: (V, S) \rightarrow S$$

$$\textit{create} :: V \rightarrow S$$

Bidirectional Laws:

$$\textit{put} (\textit{get} \ s, s) = s \quad \text{-- (GetPut)}$$

$$\textit{get} (\textit{put} \ (v, s)) = v \quad \text{-- (PutGet)}$$

$$\textit{get} (\textit{create} \ v) = v \quad \text{-- (CreateGet)}$$

# Bidirectional Transformation

A function maps source into view:

$$\textit{get} :: S \rightarrow V$$

The backward transformation:

$$\textit{put} :: (V, S) \rightarrow S$$

$$\textit{create} :: V \rightarrow S$$

Bidirectional Laws:

$$\textit{put} (\textit{get} s, s) = s \quad \text{-- (GetPut)}$$

$$\textit{get} (\textit{put} (v, s)) = v \quad \text{-- (PutGet)}$$

$$\textit{get} (\textit{create} v) = v \quad \text{-- (CreateGet)}$$

# Bidirectional Transformation

A function maps source into view:

$$\textit{get} :: S \rightarrow V$$

The backward transformation:

$$\textit{put} :: (V, S) \rightarrow S$$

$$\textit{create} :: V \rightarrow S$$

Bidirectional Laws:

$$\textit{put} (\textit{get} s, s) = s \quad \text{-- (GetPut)}$$

$$\textit{get} (\textit{put} (v, s)) = v \quad \text{-- (PutGet)}$$

$$\textit{get} (\textit{create} v) = v \quad \text{-- (CreateGet)}$$

# Bidirectional Transformation

A function maps source into view:

$$get :: S \rightarrow V$$

The backward transformation:

$$bwd :: (V, S) \rightarrow S$$

Bidirectional Laws:

$$bwd (get\ s, s) = s \quad \text{-- (GetBwd)}$$

$$get (bwd (v, s)) = v \quad \text{-- (BwdGet)}$$

# View Declaration

*to* as *get* and *from* as backward transformation

**type** *Queue a* = (*List a*, *List a*)

**view** *Queue a* = *List a*

*to* = *append*

*emptyQ* = *Nil*

*enQ* :: *a* → *Queue a* → *Queue a*

*enQ* *x* *xs* = *append xs (Cons x Nil)*

*deQ* :: *Queue a* → *Queue a*

*deQ* (*Cons x xs*) = *xs*

# Translated Views

*to* as *get* and *from* as backward transformation

**type** *Queue* *a* = (*List* *a*, *List* *a*)

**type** *QueueL* *a* = *List* *a*

*to* = *append*; *from* = *append*  $\gg$

*emptyQv* = *Nil*

*enQv* (*x*, *xs*) = *append* (*xs*, *Cons* *x Nil*)

*deQv* (*Cons* *x xs*) = *xs*

*emptyQ* = *from* (*emptyQv*,  $\perp$ )

*enQ* (*x*, *xs*) = *from* (*enQv* (*x*, *to xs*), *xs*)

*deQ* *xs* = *from* (*deQv* (*to xs*), *xs*)

# Current bidirectional transformation systems are stretched

## General datatypes

not suitable for combinator approaches

## Arbitrary updates

not suitable constant complement approaches

# Current bidirectional transformation systems are stretched

## General datatypes

not suitable for combinator approaches

## Arbitrary updates

not suitable constant complement approaches

# What can we do?

- Use complement approach to construct a partial put function
  - We have an algorithm of generating complement functions for a general purpose language
- Supplement it with a create function when the complement value is affected by updates
- **The challenge : how to construct a well-behaved and total create function?**

# What can we do?

- Use complement approach to construct a partial put function
  - We have an algorithm of generating complement functions for a general purpose language
- Supplement it with a create function when the complement value is affected by updates
- The challenge : how to construct a well-behaved and total create function?

# What can we do?

- Use complement approach to construct a partial put function
  - We have an algorithm of generating complement functions for a general purpose language
- Supplement it with a create function when the complement value is affected by updates
- **The challenge : how to construct a well-behaved and total create function?**