The worker/wrapper transformation

Jaak Randmets

April 20, 2010
Introduction

- The worker/wrapper transformation is a technique for changing the type of computation.
- Usually the aim is to improve performance by improving the choice of data structures used.
- Well known to compiler writers, but not so in the functional programming community.
- In this talk we provide systematic recipe for its use and explore it using wide range of examples.
The basic idea

Given some (recursive) function:

\[ f = \text{body} \]

the first step is to apply appropriate functions \text{wrap} and \text{unwrap} that allow the function \( f \) to be redefined by equation

\[ f = \text{wrap} (\text{unwrap body}) \]

Next step is to split the function into two by naming the intermediate result:

\[
\begin{align*}
  f & = \text{wrap work} \\
  \text{work} & = \text{unwrap body}
\end{align*}
\]

We then eliminate the mutual recursion:

\[
\begin{align*}
  f & = \text{wrap work} \\
  \text{work} & = \text{unwrap (body [\text{wrap work} / f])}
\end{align*}
\]
The worker/wrapper transformation

We begin by defining fixed point operator in Haskell:

\[
fix :: (a \to a) \to a
fix \ f = f (fix \ f)
\]

In order to formalize the worker/wrapper transformation we will use property of fixed points known as the *rolling rule*:

\[
fix (g \circ f) = g (fix (f \circ g))
\]

Intuitively this is correct because both sides expand to the application \(g (f (g (f \ldots)))\).
The worker/wrapper transformation

Supposed we have computation defined as a fixed point:

\[
\text{comp} :: A \\
\text{comp} = \text{fix } \text{body} \\
\text{body} :: A \rightarrow A
\]

We wish to change the underlying type \( A \) to some other type \( B \).

Worker/wrapper approach to this problem is to define conversion functions:

\[
\text{unwrap} :: A \rightarrow B \\
\text{wrap} :: B \rightarrow A
\]

such that \( \text{wrap} \circ \text{unwrap} = \text{id} \).
The worker/wrapper transformation

```
comp
= { applying comp }

fix body
= { id is identity for ⋄ }

fix (id ⋄ body)
= { assuming wrap ⋄ unwrap = id }

fix (wrap ⋄ unwrap ⋄ body)
= { use of rolling rule }

wrap (fix (unwrap ⋄ body ⋄ wrap))
= { define work = fix (unwrap ⋄ body ⋄ wrap) }

wrap work
```
The worker/wrapper transformation

Sometimes we might require a weaker property. Any of the following assumptions is valid:

**Worker/wrapper assumptions**

- \( \text{wrap} \circ \text{unwrap} = \text{id} \)
- \( \text{wrap} \circ \text{unwrap} \circ \text{body} = \text{body} \)
- \( \text{fix} \left( \text{wrap} \circ \text{unwrap} \circ \text{body} \right) = \text{fix} \text{ body} \)

In general it’s not the case that \( \text{unwrap} \circ \text{wrap} = \text{id} \). However, following fusion property holds:

**Worker/wrapper fusion**

If \( \text{wrap} \circ \text{unwrap} = \text{id} \), then \( \text{unwrap} \left( \text{wrap} \text{ work} \right) = \text{work} \).

Can easily be shown by taking:

\[
\text{work} = \text{unwrap} \left( \text{body} \left( \text{wrap} \text{ work} \right) \right)
\]
If $comp :: A$ is a recursive computation defined by $comp = fix \ body$ for some $body :: A \rightarrow A$, and $wrap :: B \rightarrow A$ and $unwrap :: A \rightarrow B$ are conversion functions satisfying any of the worker/wrapper assumptions, then:

$$comp = wrap \ work$$

where $work :: B$ is defined by:

$$work = fix \ (unwrap \circ body \circ wrap)$$
**Difference lists**

- Difference lists is an alternative way of representing lists:

  \[
  \textbf{type} \ DList \ a = [a] \rightarrow [a]
  \]

- Naturally we need to convert to and from difference lists:

  \[
  \begin{align*}
  \text{fromList} & \quad :: \ [a] \rightarrow \ DList \ a \\
  \text{fromList} \ \text{xs} & \quad = \ (\text{xs} \uplus \ ) \\
  \text{toList} & \quad :: \ DList \ a \rightarrow \ [a] \\
  \text{toList} \ f & \quad = \ f \ [ ]
  \end{align*}
  \]

- We observe an identity \( \text{toList} \circ \text{fromList} = \text{id} \):

  \[
  (\text{toList} \circ \text{fromList}) \ \text{xs} = \ \text{xs} \uplus \ [ ] = \ \text{xs}
  \]
Difference lists

- Important property of difference lists is that \texttt{fromList} forms a morphism from lists to functions:

\[
\text{fromList} \, (xs \oplus ys) = \text{fromList} \, xs \circ \text{fromList} \, ys \\
\text{fromList} \, [] = id
\]

- We can verify this by simple calculations:

\[
\text{fromList} \, (xs \oplus ys) \, zs = (xs \oplus ys) \oplus zs \\
= xs \oplus (ys \oplus zs) \\
= \text{fromList} \, xs \, (ys \oplus zs) \\
= \text{fromList} \, xs \, (\text{fromList} \, ys \, zs) \\
= (\text{fromList} \, xs \circ \text{fromList} \, ys) \, zs
\]

\[
\text{fromList} \, [] \, zs = [] \oplus zs \\
= zs
\]
Reverse

Let us consider the following definition of reverse:

\[
rev :: [a] \rightarrow [a] \\
rev [] = [] \\
rev (x : xs) = rev xs ++ [x]
\]

As a first step we redefine \(rev\) as a fixed point

\[
rev :: [a] \rightarrow [a] \\
rev = fix body \\
body :: ([a] \rightarrow [a]) \rightarrow ([a] \rightarrow [a]) \\
body f [] = [] \\
body f (x : xs) = f xs ++ [x]
\]
Reverse

- Our aim is to change to computation type \([a] \to [a]\) to a type \([a] \to DList a\):

\[
unwrap \quad :: \quad ([a] \to [a]) \to ([a] \to DList a)
\]
\[
unwrap \ f = fromList \circ f
\]
\[
wrap \quad :: \quad ([a] \to DList a) \to ([a] \to [a])
\]
\[
wrap \ g = toList \circ g
\]

- We can verify the worker/wrapper assumption by:

\[
(wrap \circ unwrap) \ f = wrap (unwrap \ f)
\]
\[
= toList \circ unwrap \ f
\]
\[
= toList \circ fromList \circ f
\]
\[
= id \circ f
\]
\[
= f
\]
Simplifying \( rev \)

\[
\begin{align*}
rev & :: \ [a] \rightarrow [a] \\
rev & = \text{wrap work} \\
work & :: \ [a] \rightarrow \text{DList } a \\
work & = \text{unwrap (body (wrap work))}
\end{align*}
\]

- Inline \( \text{wrap} \) in body of \( rev \).
- \( \eta \)-expand \( work \).
Reverse

Simplifying \( \text{rev} \)

\[
\begin{align*}
\text{rev} & :: [a] \rightarrow [a] \\
\text{rev } xs & = \text{work } xs \ [] \\
\text{work} & :: [a] \rightarrow DList a \\
\text{work } xs & = \text{unwrap } (\text{body } (\text{wrap } \text{work})) \ xs
\end{align*}
\]

- Expand \( \text{unwrap} \).
Reverse

Simplifying \( \text{rev} \)

\[
\begin{align*}
\text{rev} &\:: [a] \rightarrow [a] \\
\text{rev} \; xs &\; = \; \text{work} \; xs \; [] \\
\text{work} &\:: [a] \rightarrow \text{DLList} \; a \\
\text{work} \; xs &\; = \; \text{fromList} \; (\text{body} \; (\text{wrap} \; \text{work}) \; xs)
\end{align*}
\]

- Inline \( \text{body} \).
Reverse

Simplifying \textit{rev}

\begin{align*}
  \text{rev} & : [a] \rightarrow [a] \\
  \text{rev} \; xs & = \text{work} \; xs \; [] \\
  \text{work} & : [a] \rightarrow \text{DList} \; a \\
  \text{work} \; xs & = \text{fromList} \; (\text{case} \; xs \; \text{of} \\
  & \quad [] \quad \rightarrow \quad [] \\
  & \quad (x : xs) \quad \rightarrow \quad \text{wrap} \; \text{work} \; xs \; +\; [x])
\end{align*}

- case transformation.
Reverse

Simplifying \( rev \)

\[
\begin{align*}
rev & : [a] \rightarrow [a] \\
rev \ xs & = work \ xs \ [] \\
work & : [a] \rightarrow DList \ a \\
work \ [] & = fromList \ [] \\
work \ (x : xs) & = fromList \ (wrap \ work \ xs \ ++ \ [x])
\end{align*}
\]

- \( fromList \) is morphism.
Reverse

Simplifying \( rev \)

\[
rev \quad :: \quad [a] \to [a] \\
rev \hspace{0.1cm}xs \quad = \quad work \hspace{0.1cm}xs \hspace{0.1cm}[] \\
work \quad :: \quad [a] \to DList \hspace{0.1cm}a \\
work \hspace{0.1cm}[] \quad = \quad id \\
work \hspace{0.1cm}(x : xs) \quad = \quad fromList \hspace{0.1cm}(\text{wrap} \hspace{0.1cm}work \hspace{0.1cm}xs) \circ fromList \hspace{0.1cm}[x]
\]

\[
\quad \text{fromList} \hspace{0.1cm}(\text{wrap} \hspace{0.1cm}work \hspace{0.1cm}xs) \\
= \quad \text{unwrap} \hspace{0.1cm}(\text{wrap} \hspace{0.1cm}work) \hspace{0.1cm}xs \\
= \quad work \hspace{0.1cm}xs
\]
Reverse

Simplifying \texttt{rev}

\[
\begin{align*}
\texttt{rev} & \quad :: \ [a] \rightarrow [a] \\
\texttt{rev} \ \texttt{xs} & \quad = \ \texttt{work} \ \texttt{xs} \ [\ [] \\
\texttt{work} & \quad :: \ [a] \rightarrow \textit{DList} \ a \\
\texttt{work} \ [\ [] & \quad = \ \texttt{id} \\
\texttt{work} \ (x : \texttt{xs}) & \quad = \ \texttt{work} \ \texttt{xs} \ \circ \ \textit{fromList} \ [x]
\end{align*}
\]

- \(\eta\)-expand \texttt{work}.
- Expand \textit{fromList} and \(\circ\).
Reverse

Simplifying \textit{rev}

\[
\begin{align*}
\text{rev} & : [a] \rightarrow [a] \\
\text{rev } xs & = \text{work } xs \ [\ ] \\
\text{work} & : [a] \rightarrow [a] \rightarrow [a] \\
\text{work } [] \ ys & = ys \\
\text{work } (x : xs) \ ys & = \text{work } xs \ (x : ys)
\end{align*}
\]

- We have reached linear time accumulating version of list reversing function.
Memoisation

Our approach to memoising is to observe that any function $f$ from natural numbers can be represented as infinite stream $[f \ 0, f \ 1, ...]$.

unwrap/wrap

\[
\begin{align*}
\text{unwrap} & \quad :: (Nat \to a) \to Stream a \\
\text{unwrap } f & \quad = \text{map } f \ [0 \ ..] \\
& \quad = f \ 0 : \text{unwrap } (f \circ (+1)) \\
\text{wrap} & \quad :: Stream a \to (Nat \to a) \\
\text{wrap } xs & \quad = (xs!!)
\end{align*}
\]

\[
\begin{align*}
\text{ (!!)} & \quad :: Stream a \to Nat \to a \\
x \!\! \! 0 & \quad = \text{head } xs \\
x \!\! (n + 1) & \quad = (\text{tail } xs) \!\! n
\end{align*}
\]
Memoisation

We will show that $\text{wrap} \circ \text{unwrap} = \text{id}$ by expanding $\text{wrap}$ and making arguments explicit: $(\text{unwrap } f) !! n = f \ n$.

- **Base case:**
  \[
  (\text{unwrap } f) !! 0 \\
  = (f \ 0 : \text{unwrap } (f \circ (+1))) !! 0 \\
  = f \ 0
  \]

- **Inductive case:**
  \[
  (\text{unwrap } f) !! (n + 1) \\
  = (f \ 0 : \text{unwrap } (f \circ (+1))) !! (n + 1) \\
  = \text{unwrap } (f \circ (+1)) !! n \\
  = (f \circ (+1)) \ n \\
  = f \ (n + 1)
  \]
Fibonacci function

\[
\begin{align*}
\text{fib} & \quad :: \quad \text{Nat} \rightarrow \text{Nat} \\
\text{fib} & \quad = \quad \text{fix body} \\
\text{body} & \quad :: \quad (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{body} \ f \ 0 & \quad = \quad 0 \\
\text{body} \ f \ 1 & \quad = \quad 1 \\
\text{body} \ f \ (n + 2) & \quad = \quad f \ n + f \ (n + 1)
\end{align*}
\]
Fibonacci

Simplifying fib

\[
\begin{align*}
\text{fib} & \quad :: \quad \text{Nat} \rightarrow \text{Nat} \\
\text{fib} & \quad = \quad \text{wrap } \text{work} \\
\text{work} & \quad :: \quad \text{Stream} \ \text{Nat} \\
\text{work} & \quad = \quad \text{unwrap } (\text{body } (\text{wrap } \text{work}))
\end{align*}
\]

- Apply \text{wrap} in \text{fib}.
- Apply \text{unwrap} in \text{work}.
Fibonacci

Simplifying \( \text{fib} \)

\[
\begin{align*}
\text{fib} &:: \text{Nat} \to \text{Nat} \\
\text{fib } n & = \text{work } !! \ n \\
\text{work} &:: \text{Stream Nat} \\
\text{work} & = \text{map } (\text{body } (\text{wrap work})) \ [0..]
\end{align*}
\]

- Inline \( \text{body} \).
Fibonacci

Simplifying \( \text{fib} \)

\[
\text{fib} :: \text{Nat} \rightarrow \text{Nat} \\
\text{fib } n = \text{work } !! n \\
\text{work} :: \text{Stream Nat} \\
\text{work} = \text{map} (\lambda n \rightarrow \text{case } n \\text{ of} \\
0 \rightarrow 0 \\
1 \rightarrow 1 \\
(n + 2) \rightarrow \text{wrap work } n + \text{wrap work } (n + 1)) [0 ..]
\]

- Apply \( \text{wrap} \).
Simplifying \( \text{fib} \)

\[
\begin{align*}
\text{fib} & \quad : \quad \text{Nat} \rightarrow \text{Nat} \\
\text{fib} \ n & \quad = \quad \text{work} \ !! \ n \\
\text{work} & \quad : \quad \text{Stream Nat} \\
\text{work} & \quad = \quad \text{map} \ (\lambda n \rightarrow \text{case} \ n \ \text{of} \\
0 & \quad \rightarrow \quad 0 \\
1 & \quad \rightarrow \quad 1 \\
(n + 2) & \quad \rightarrow \quad \text{work} \ !! \ n + \text{work} \ !! \ (n + 1)) \ [0..]
\end{align*}
\]

Finally introduce \( f \).
We have reached a quadratic Fibonacci function from exponential one.
Any type can be alternatively represented as a continuation. Idea is to represent value \( x \) as a function \( \lambda c \rightarrow c \times \) that takes a (continuation) \( c \) and applies it to \( x \).

\[
\text{type } \textit{Cont } a = (a \rightarrow a) \rightarrow a
\]

We can convert from and to continuations as follows:

\[
to\textit{Cont} \quad ::\quad a \rightarrow \textit{Cont } a
\]

\[
to\textit{Cont} x = \lambda c \rightarrow c \times
\]

\[
\textit{fromCont} \quad ::\quad \textit{Cont } a \rightarrow a
\]

\[
\textit{fromCont } f = f \ id
\]

It’s easy to show that:

\[
(from\textit{Cont} \circ to\textit{Cont}) x = (to\textit{Cont} x) \ id
\]

\[
= (\lambda c \rightarrow c \times) \ id
\]

\[
= x
\]
Evaluation

We will now consider simple expression language:

```
data Expr = Val Int
         | Expr ⊕ Expr
         | Throw
         | Catch Expr Expr
```

With standard evaluation function:

```
eval :: Expr → MInt
eval (Val n) = Just n
eval (e₀ ⊕ e₁) = case eval e₀ of
  Nothing → Nothing
  Just n → case eval e₁ of
            Nothing → Nothing
            Just m → Just (n + m)
eval Throw = Nothing
eval (Catch e₀ e₁) = case eval e₀ of
  Nothing → eval e₁
  Just n → Just n
```
Evaluation

- One might expect to move to representation of type $\text{Expr} \rightarrow \text{Cont MInt}$. Instead we will have different continuation for exceptional control flow and regular control flow.

- We split $\text{MInt} \rightarrow \text{MInt}$ into two and reach type:

$$
\text{Expr} \rightarrow (\text{Int} \rightarrow \text{MInt}) \rightarrow \text{MInt} \rightarrow \text{MInt}
$$

- Wrap/unwrap are given by:

\[
\text{unwrap } g \ e \ s \ f = \text{case } g \ e \text{ of}
\]
\[
\text{Nothing } \rightarrow f \\
\text{Just } n \rightarrow s \ n
\]

\[
\text{wrap } h \ e = h \ e \text{ Just Nothing}
\]
Evaluation

- Worker/wrapper assumption can easily verified as follows:

\[(\text{wrap} \circ \text{unwrap}) \; g \; e\]
\[= \text{wrap} \; (\text{unwrap} \; g) \; e\]
\[= \text{unwrap} \; g \; e \; \text{Just Nothing}\]
\[= \text{case} \; g \; e \; \text{of}\]
\[\text{Nothing} \rightarrow \text{Nothing}\]
\[\text{Just} \; n \rightarrow \text{Just} \; n\]
\[= g \; e\]

- We can now apply the worker/wrapper transformation.
Evaluation

Simplifying \textit{eval}

\begin{align*}
\text{eval} & : \text{Expr} \rightarrow \text{MInt} \\
\text{eval} & = \text{wrap work} \\
\text{work} & : \text{Expr} \rightarrow (\text{Int} \rightarrow \text{MInt}) \rightarrow \text{MInt} \rightarrow \text{MInt} \\
\text{work} & = \text{unwrap (body (wrap work))}
\end{align*}

- Inline \textit{wrap}.
- \(\eta\)-expand \textit{work}.
Simplifying `eval`

\[
\begin{align*}
\text{eval} & \quad :: \quad \text{Expr} \rightarrow \text{MInt} \\
\text{eval} & \quad = \quad \text{work} \; e \; \text{Just} \; \text{Nothing} \\
\text{work} & \quad :: \quad \text{Expr} \rightarrow (\text{Int} \rightarrow \text{MInt}) \rightarrow \text{MInt} \rightarrow \text{MInt} \\
\text{work} \; e \; s \; f & \quad = \quad \text{unwrap} \; (\text{body} \; (\text{wrap} \; \text{work})) \; e \; s \; f
\end{align*}
\]

- Apply `unwrap`. 

Simplifying `eval`

```
eval :: Expr -> MInt

eval = work e Just Nothing

work :: Expr -> (Int -> MInt) -> MInt -> MInt

work e s f = case body (wrap work) e of
  Nothing -> f
  Just n -> s n
```

- Apply `body`.
Simplifying eval

\[
\text{work } e \, s \, f = \text{case } (\text{case } e \text{ of } \\
\quad \text{Val } n \quad \rightarrow \text{Just } n \\
\quad e_0 \oplus e_1 \quad \rightarrow \text{case } \text{wrap work } e_0 \text{ of } \\
\quad \text{Nothing} \quad \rightarrow \text{Nothing} \\
\quad \text{Just } n \quad \rightarrow \text{case } \text{wrap work } e_1 \text{ of } \\
\quad \text{Nothing} \quad \rightarrow \text{Nothing} \\
\quad \text{Just } m \quad \rightarrow \text{Just } (n + m) \\
\quad \text{Throw} \quad \rightarrow \text{Nothing} \\
\quad \text{Catch } e_0 \, e_1 \quad \rightarrow \text{case } \text{wrap work } e_0 \text{ of } \\
\quad \text{Nothing} \quad \rightarrow \text{wrap work } e_1 \\
\quad \text{Just } n \quad \rightarrow \text{Just } n) \text{ of } \\
\quad \text{Nothing} \quad \rightarrow f \\
\quad \text{Just } n \quad \rightarrow s \, n
\]
Evaluation

Simplifying `eval`

\[
\text{work } e \ s \ f = \text{case } e \ \text{of}
\]
\[
\begin{align*}
\text{Val } n & \rightarrow s \ n \\
\text{Val } n \oplus \text{Val } n & \rightarrow \text{case } \text{wrap work } e_0 \ \text{of} \\
\text{Nothing} & \rightarrow f \\
\text{Just } n & \rightarrow \text{case } \text{wrap work } e_1 \ \text{of} \\
\text{Nothing} & \rightarrow f \\
\text{Just } m & \rightarrow s (n + m) \\
\text{Throw} & \rightarrow f \\
\text{Catch } e_0 \ e_1 & \rightarrow \text{case } \text{wrap work } e_0 \ \text{of} \\
\text{Nothing} & \rightarrow \text{case } \text{wrap work } e_1 \ \text{of} \\
\text{Nothing} & \rightarrow f \\
\text{Just } n & \rightarrow s \ n \\
\text{Just } n & \rightarrow s \ n
\end{align*}
\]
Evaluation

Simplifying eval

\[
\begin{align*}
work (\text{Val } n) \ s \ f & = s \ n \\
work (e_0 \oplus e_1) \ s \ f & = \text{case } \text{wrap work } e_0 \ \text{of} \\
 & \text{Nothing } \rightarrow f \\
 & \text{Just } n \rightarrow \text{case } \text{wrap work } e_1 \ \text{of} \\
 & \qquad \text{Nothing } \rightarrow f \\
 & \qquad \text{Just } m \rightarrow s (n + m) \\
work \ \text{Throw} \ s \ f & = f \\
work (\text{Catch } e_0 \ e_1) \ s \ f & = \text{case } \text{wrap work } e_0 \ \text{of} \\
 & \text{Nothing } \rightarrow \text{case } \text{wrap work } e_1 \ \text{of} \\
 & \qquad \text{Nothing } \rightarrow f \\
 & \qquad \text{Just } n \rightarrow s \ n \\
& \text{Just } n \rightarrow s \ n
\end{align*}
\]
By re-exposing *unwrap* we can perform further simplification:

\[
\text{case } \text{wrap work } x \text{ of}
\]
\[Nothing \rightarrow g
\]
\[Just \ n \rightarrow s \ n
\]
\[= \ \{ \text{unapply } \text{unwrap} \ \}\]
\[\text{unwrap} \ (\text{wrap work}) \times s \ g
\]
\[= \ \{ \text{worker/wrapper fusion} \ \}\]
\[\text{work} \times s \ g
\]

We can apply this multiple times.
## Evaluation

### Simplifying `eval`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>work (Val n) s f</code></td>
<td><code>= s n</code></td>
</tr>
<tr>
<td><code>work (e_0 ⊕ e_1) s f</code></td>
<td><code>= work e_0 (\lambda n \rightarrow\&lt;br&gt;work e_1 (\lambda m \rightarrow s (n + m)) f) f</code></td>
</tr>
<tr>
<td><code>work Throw s f</code></td>
<td><code>= f</code></td>
</tr>
<tr>
<td><code>work (Catch e_0 e_1) s f</code></td>
<td><code>= work e_0 s (work e_1 s f)</code></td>
</tr>
</tbody>
</table>

- Corresponds to abstract machine that works on two stacks, one for normal evaluation and other for handling exceptions.
Fin.