The Worker/Wrapper Transformation

Andy Gill\textsuperscript{1}  Graham Hutton\textsuperscript{2}

\textsuperscript{1}Galois, Inc.
\textsuperscript{2}University of Nottingham

March 11, 2008
The Worker/Wrapper Transformation is a rewrite technique which changes the type of a (recursive) computation.

- Worker/wrapper has been used inside the Glasgow Haskell compiler since its inception to rewriting functions that use lifted values (thunks) into equivalent and more efficient functions that use unlifted values.
- This talk will explain why worker/wrapper works!
- Much, much more general that just exploiting strictness analysis.
- Worker/wrapper is about changing types.
Changing the type of a computation 

- is pervasive in functional programming
- useful in practice
- the essence of turning a specification into an implementation

Thesis:

- The Worker/Wrapper Transformation is a great technique for changing types

This talk

- Examples of what worker/wrapper can do
- Formalize the Worker/Wrapper Transformation (why it works)
- Give a recipe for its use
- Show how to apply our worker/wrapper recipe to some examples
**Example 1: Strictness Exploitation**

**Before**

```haskell
fac :: Int -> Int -> Int
fac n m = if n == 1
  then m
  else fac (n - 1) (m * n)
```

- $n$ is trivially strict, $m$ is provably strict
- Can use $\text{Int#}$, a strict version of Int that is passed by value for $n$ and $m$

**After**

```haskell
fac n m = box (work (unbox n) (unbox m))

work :: Int# -> Int# -> Int#
work n# m# = if n# ==# 1#
  then m#
  else work (n# -# 1#) (m# *# n#)
```
Example 2: Avoiding Needless Deconstruction

Before

```haskell
last :: [a] -> a
last [] = error "last: []"
last (x:[]) = x
last (x:xs) = last xs
```

- The recursive call of `last` never happens with an empty list
- Subsequent recursive invocations performs a needless check for an empty list

After

```haskell
last [] = error "last: []"
last (x:xs) = work x xs

work :: a -> [a] -> a
work x [] = x
work x (y:ys) = work y ys
```
Example 3: Efficient nub

Before

nub :: [Int] -> [Int]
nub [] = []
nub (x:xs) = x : nub (filter (\y -> not (x == y)) xs)

- `filter` is applied to the tail of the argument list on each recursive call, to avoid duplication.
- It would be more efficient to remember the elements that have already been issued.

After

nub :: [Int] -> [Int]
nub xs = work xs empty

work :: [Int] -> Set Int -> [Int]
work xs except =
  case dropWhile (\x -> x `member` except) xs of
  [] -> []
  (x:xs) -> x : work xs (insert x except)
Example 4: Memoization

**Before**

```haskell
fib :: Nat -> Nat
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

- Memoization is a well-known optimization for `fib`
- Memoization is just a change in representation over the recursive call

**After**

```haskell
fib :: Nat -> Nat
fib n = work !! n

work :: [Nat]
work = map f [0..]
    where f = if n < 2 then 1 else work !! (n-1) + work !! (n-2)
```
Example 5: Double-barreled CPS Translation

Before

\[
\begin{align*}
\text{eval :: Expr} & \rightarrow \text{Maybe Int} \\
\text{eval (Val n)} & = \text{Just n} \\
\text{eval (Add x y)} & = \text{case eval x of Nothing \rightarrow Nothing} \\
& \quad \text{Just n \rightarrow case eval y of Nothing \rightarrow Nothing} \\
& \quad \text{Just m \rightarrow Just (n+m)} \\
\text{eval (Throw)} & = \text{Nothing} \\
\text{eval (Catch x y)} & = \text{case eval x of Nothing \rightarrow eval y} \\
& \quad \text{Just n \rightarrow Just n}
\end{align*}
\]

- CPS changes the result type from \(A\) to \((A \rightarrow X) \rightarrow X\)
- Again, just a change in representation

After

\[
\begin{align*}
\text{eval e} & = \text{work e Just Nothing} \\
\text{work :: Expr} & \rightarrow (\text{Int} \rightarrow \text{Maybe Int}) \rightarrow \text{Maybe Int} \rightarrow \text{Maybe Int} \\
\text{work (Val n)} & = \text{sf} = s \ n \\
\text{work (Add x y)} & = \text{work x (\n \rightarrow work y (\m \rightarrow s (n+m)) f) f} \\
\text{work (Throw)} & = \text{sf} = f \\
\text{work (Catch x y)} & = \text{work x s (work y s f)}
\end{align*}
\]
Changing the *representation* of a computation . . .

- is pervasive in functional programming
- useful in practice
- the essence of turning a specification into an implementation
- is what worker/wrapper does
Creating Workers and Wrappers for last

last :: [a] -> a
last = \ v -> case v of
    []    -> error "last: []"
    (x:xs) -> case xs of
            []    -> x
            (_:_ ) -> last xs
Creating Workers and Wrappers for last

last :: [a] -> a
last =

last_work :: a -> [a] -> a
last_work = \ x xs ->
  (\ v -> case v of
    [] -> error "last: []"
    (x:xs) -> case xs of
      [] -> x
      (_:_) -> last xs) (x:xs)

- Create the worker out of the body and an invented coercion to the target type
Creating Workers and Wrappers for last

```haskell
last :: [a] -> a
last = \ v -> case v of
    []     -> error "last: []"
    (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
    (\ v -> case v of
        []     -> error "last: []"
        (x:xs) -> case xs of
            []     -> x
            (_:_  ) -> last xs) (x:xs)
```

- Create the worker out of the body and an invented coercion to the target type
- Invent the wrapper which call the worker
Creating Workers and Wrappers for \texttt{last}

\begin{verbatim}
last :: [a] \rightarrow a
last = \ v \rightarrow case v of
   []     \rightarrow error "last: []"
   (x:xs) \rightarrow last_work x xs

last_work :: a \rightarrow [a] \rightarrow a
last_work = \ x xs \rightarrow
   (\ v \rightarrow case v of
      []     \rightarrow error "last: []"
      (x:xs) \rightarrow case xs of
         []     \rightarrow x
         (_:_   \rightarrow last xs) (x:xs)
\end{verbatim}

- Create the worker out of the body and an invented coercion to the target type
- Invent the wrapper which call the worker
- These functions are mutually recursive
last :: [a] -> a
last = \ v -> case v of
        []         -> error "last: []"
        (x:xs)     -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
            (\ v -> case v of
                     []         -> error "last: []"
                     (x:xs)     -> case xs of
                        []         -> x
                        (_:_ )     -> last xs) (x:xs)

- We now inline last inside last_work
We now inline `last` inside `last_work`

- `last_work` is now trivially recursive.
Simplify work

last :: [a] -> a
last = \ v -> case v of
        []   -> error "last: []"
        (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
            (\ v -> case v of
                []   -> error "last: []"
                (x:xs) -> case xs of
                    []   -> x
                    (_:_ ) ->
                        (\ v -> case v of
                            []   -> error "last: []"
                            (x:xs) -> last_work x xs) xs) (x:xs)

We now simplify the worker
We now simplify the worker.

Reaching our efficient implementation.
From a recursive function, construct two new functions

- **Wrapper**
  - Replacing the original function
  - Coerces call to Worker

- **Worker**
  - Performs main computation
  - Syntactically contains the body of the original function
  - Coerces call from Wrapper

The initial worker and wrapper are mutually recursive
We then inline the wrapper inside the worker, and simplify
We end up with
  - An efficient recursive worker
  - An impedance matching non-recursive wrapper
Questions about the Worker/Wrapper Transformation

- Is the technique actually correct?
- How can this be proved?
- Under what conditions does it hold?
- How should it be used in practice?
wrap and unwrap

\[
\text{last} :: [a] \rightarrow a
\]

\[
\text{work} :: a \rightarrow [a] \rightarrow a
\]
wrap and unwrap in General

\[
\text{comp} :: \ A \ \\
\text{work} :: \ B
\]

unwrap

wrap
Prerequisites

comp :: A
comp = fix body for some body :: A → A

wrap :: B → A is a coercion from type B to A
unwrap :: A → B is a coercion from type A to B

wrap ⋅ unwrap = id_A \ (basic worker/wrapper assumption)

Derivation

comp = fix body
Prerequisites

comp :: A
comp = fix body for some body :: A → A

wrap :: B → A is a coercion from type B to A
unwrap :: A → B is a coercion from type A to B

wrap ⋅ unwrap = id_A  (basic worker/wrapper assumption)

Derivation

comp = fix body
= { id is the identity for ⋅ }  
comp = fix (id ⋅ body)
Prerequisites

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

\[ \text{wrap :: } B \rightarrow A \text{ is a coercion from type } B \text{ to } A \]
\[ \text{unwrap :: } A \rightarrow B \text{ is a coercion from type } A \text{ to } B \]

\[ \text{wrap} \cdot \text{unwrap} = \text{id}_A \quad (\text{basic worker/wrapper assumption}) \]

Derivation

\[ \text{comp} = \text{fix body} \]
\[ = \{ \text{id is the identity for } \cdot \} \]
\[ \text{comp} = \text{fix (id } \cdot \text{ body)} \]
\[ = \{ \text{assuming wrap} \cdot \text{unwrap} = \text{id } \} \]
\[ \text{comp} = \text{fix (wrap} \cdot \text{unwrap } \cdot \text{body)} \]
Prerequisites

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

\[ \text{wrap :: } B \rightarrow A \text{ is a coercion from type } B \text{ to } A \]
\[ \text{unwrap :: } A \rightarrow B \text{ is a coercion from type } A \text{ to } B \]

\[ \text{wrap} \cdot \text{unwrap} = \text{id}_A \quad (\text{basic worker/wrapper assumption}) \]

Derivation

\[ \text{comp} = \text{fix body} \]
\[ = \{ \text{id is the identity for } \cdot \} \]
\[ \text{comp} = \text{fix (id } \cdot \text{ body)} \]
\[ = \{ \text{assuming wrap} \cdot \text{unwrap} = \text{id} \} \]
\[ \text{comp} = \text{fix (wrap } \cdot \text{unwrap } \cdot \text{ body)} \]
\[ = \{ \text{rolling rule } \} \]
\[ \text{comp} = \text{wrap (fix (unwrap } \cdot \text{body } \cdot \text{ wrap))} \]
Prerequisites

\[ \text{comp} :: A \]
\[ \text{comp} = \text{fix body for some body} :: A \rightarrow A \]
\[ \text{wrap} :: B \rightarrow A \text{ is a coercion from type } B \text{ to } A \]
\[ \text{unwrap} :: A \rightarrow B \text{ is a coercion from type } A \text{ to } B \]
\[ \text{wrap} \cdot \text{unwrap} = \text{id}_A \quad \text{(basic worker/wrapper assumption)} \]

Derivation

\[ \text{comp} = \text{fix body} \]
\[ = \{ \text{id is the identity for } \cdot \} \]
\[ \text{comp} = \text{fix (id } \cdot \text{ body)} \]
\[ = \{ \text{assuming wrap} \cdot \text{unwrap} = \text{id} \} \]
\[ \text{comp} = \text{fix (wrap} \cdot \text{unwrap} \cdot \text{body)} \]
\[ = \{ \text{rolling rule} \} \]
\[ \text{comp} = \text{wrap (fix (unwrap} \cdot \text{body} \cdot \text{wrap))} \]
\[ = \{ \text{define work} = \text{fix (unwrap} \cdot \text{body} \cdot \text{wrap)} \} \]
\[ \text{comp} = \text{wrap work} \]
\[ \text{work} = \text{fix (unwrap} \cdot \text{body} \cdot \text{wrap)} \]
Prerequisites

\[ \text{comp} :: A \]
\[ \text{comp} = \text{fix body for some body} :: A \to A \]

\[ \text{wrap} :: B \to A \text{ is a coercion from type } B \text{ to } A \]
\[ \text{unwrap} :: A \to B \text{ is a coercion from type } A \text{ to } B \]

\[ \text{wrap} \cdot \text{unwrap} = \text{id}_A \]

(basic worker/wrapper assumption)

Worker/Wrapper Theorem

If the above prerequisites hold, then

\[ \text{comp} = \text{fix body} \]

can be rewritten as

\[ \text{comp} = \text{wrap work} \]

where \( \text{work} :: B \) is defined by

\[ \text{work} = \text{fix (unwrap} \cdot \text{body} \cdot \text{wrap)} \]
The Worker/Wrapper Assumptions

Key step of proof

\[ \text{fix } (\text{id } \cdot \text{ body}) \]

\[ = \{ \text{assuming } \text{wrap } \cdot \text{unwrap } = \text{id} \} \]

\[ \text{fix } (\text{wrap } \cdot \text{unwrap } \cdot \text{body}) \]

We can actually use any of three different assumptions here

\[ \text{wrap } \cdot \text{unwrap } = \text{id} \]  \hspace{1cm} \text{(basic assumption)}

\[ \Downarrow \]

\[ \text{wrap } \cdot \text{unwrap } \cdot \text{body } = \text{body} \]  \hspace{1cm} \text{(body assumption)}

\[ \Downarrow \]

\[ \text{fix } (\text{wrap } \cdot \text{unwrap } \cdot \text{body} ) = \text{fix } \text{body} \]  \hspace{1cm} \text{(fix-point assumption)}
The Worker/Wrapper Recipe

**Recipe**

- Express the computation as a least fixed point;
- Choose the desired new type for the computation;
- Define conversions between the original and new types;
- Check they satisfy one of the worker/wrapper assumptions;
- Apply the worker/wrapper transformation;
- Simplify the resulting definitions.

We simplify to remove the overhead of the wrap and unwrap coercions, often using fusion, including the worker/wrapper fusion property.

**The Worker/Wrapper Fusion Property**

\[
\text{If wrap} \cdot \text{unwrap} = \text{id, then (unwrap} \cdot \text{wrap) work} = \text{work}
\]
Creating Workers and Wrappers for last

\[
\text{last} :: [a] \rightarrow a
\]
\[
\text{work} :: a \rightarrow [a] \rightarrow a
\]

\[
\text{wrap} \quad \text{fn} = \lambda \text{xs} \rightarrow \text{case xs of}
\]
\[
[\] \rightarrow \text{error "last: []"}
\]
\[
(x:xs) \rightarrow \text{fn} \ x \ x s
\]

\[
\text{unwrap} \quad \text{fn} = \lambda \ x \ x s \rightarrow \text{fn} \ (x:xs)
\]

\[
\text{last} = \text{fix body}
\]

\[
\text{body last} = \lambda v \rightarrow \text{case v of}
\]
\[
[\] \rightarrow \text{error "last: []"}
\]
\[
(x:[]) \rightarrow x
\]
\[
(x:x$s) \rightarrow \text{last} \ x s
\]

Andy Gill, Graham Hutton
Testing the basic worker/wrapper assumption: Does $\text{wrap} \cdot \text{unwrap} = \text{id}$?

\[
\text{wrap} \cdot \text{unwrap} \\
= \{ \text{apply wrap, unwrap and } \cdot \} \\
\lambda \text{fn} \rightarrow \\
\quad (\lambda \text{xs} \rightarrow \text{case xs of} \\
\quad \quad \quad [] \rightarrow \text{error "last: []"} \\
\quad \quad \quad (x:xs) \rightarrow (\lambda x \text{ xs} \rightarrow \text{fn}(x:xs)) \ x \ \text{xs}) \\
= \{ \beta\text{-reduction} \} \\
\lambda \text{fn} \rightarrow \\
\quad (\lambda \text{xs} \rightarrow \text{case xs of} \\
\quad \quad \quad [] \rightarrow \text{error "last: []"} \\
\quad \quad \quad (x:xs) \rightarrow \text{fn}(x:xs))
\]

Clearly not equal to $\text{id} :: ([a] \rightarrow a) \rightarrow ([a] \rightarrow a)$
Testing the body worker/wrapper assumption:  
Does wrap \cdot unwrap \cdot body = body?

\[
\text{wrap} \cdot \text{unwrap} \cdot \text{body} \\
= \{ \text{apply wrap, unwrap and } \cdot \} \\
(\text{\textbackslash } \text{fn } \rightarrow \\
(\text{\textbackslash } \text{xs } \rightarrow \text{case xs of} \\
[] \rightarrow \text{error } "\text{last: []}" \\
(x:xs) \rightarrow (\text{\textbackslash } x \ xs \rightarrow \text{fn (x:xs)) x xs}) \\
(\text{\textbackslash } \text{last } v \rightarrow \text{case v of} \\
[] \rightarrow \text{error } "\text{last: []}" \\
(x:[]) \rightarrow x \\
(x:xs) \rightarrow \text{last } xs)
\]
Testing the body worker/wrapper assumption:
Does \( \text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body} \)?
Testing the body worker/wrapper assumption:
Does \( \text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body} \)?

\[
\text{wrap} \cdot \text{unwrap} \cdot \text{body}
\]
\[
= \{ \text{apply \ wrap, \ unwrap \ and} \ \cdot \ \}\n\]
\[
= \{ \beta\text{-reductions} \} \n\]
\[
= \{ \text{case of known constructors} \} \n\]

\[
(\ \text{fn} \rightarrow
\quad (\ \text{x} \rightarrow \text{case} \ \text{x} \ \text{of}
\quad \quad [\ ] \rightarrow \text{error} \ "\text{last: [\]}"
\quad \quad (\text{x}:\text{xs}) \rightarrow \text{case} \ \text{x} \ \text{of}
\quad \quad \quad \quad [\ ] \rightarrow \text{x}
\quad \quad \quad \quad \text{xs} \rightarrow \text{fn} \ \text{x} ))
\]
Testing the body worker/wrapper assumption: Does \( \text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body} \)?

\[
\text{wrap} \cdot \text{unwrap} \cdot \text{body} \\
= \{ \text{apply wrap, unwrap and } \cdot \} \\
= \{ \beta\text{-reductions } \} \\
= \{ \text{case of known constructors } \} \\
= \{ \text{common up case } \}
\]

\[
(\lambda \text{fn} \rightarrow \\
 (\lambda \text{xs} \rightarrow \text{case xs of} \\
 [ ] \rightarrow \text{error } "\text{last: } []" \\
 (x:[]) \rightarrow x \\
 (x:xs) \rightarrow \text{fn } xs))
\]

Which equals \( \text{body} \). QED.
Applying the Worker/Wrapper Transformation

Before

\[
\text{last} = \text{fix body}
\]

\[
\text{last} :: [a] \rightarrow a
\]

\[
\text{work} :: a \rightarrow [a] \rightarrow a
\]

\[
\text{work} = \text{fix ( unwrap . body . wrap )}
\]
last :: [a] -> a
last xs = case xs of
  [] -> error "last: []"
  (x:xs) -> work x xs

work :: a -> [a] -> a
work = fix ( \ fn x xs -> fn (x:xs))
  . ( \ last v -> case v of
      [] -> error "last: []"
      (x:[]) -> x
      (x:xs) -> last xs)
  . ( \ fn xs -> case xs of
      [] -> error "last: []"
      (x:xs) -> fn x xs)
)
last :: [a] -> a
last xs = case xs of
  [] -> error "last: []"
  (x:xs) -> work x xs

work :: a -> [a] -> a
work = fix ( \ fn x xs ->
  case (x:xs) of
    [] -> error "last: []"
    (x:[]) -> x
    (x:xs) -> case xs of
      [] -> error "last: []"
      (x:xs) -> fn x xs
  )
last :: [a] -> a
last xs = case xs of
    [] -> error "last: []"
    (x:xs) -> work x xs

work :: a -> [a] -> a
work = fix ( \ fn x xs ->
    case xs of
    [] -> x
    xs -> case xs of
    [] -> error "last: []"
    (x:xs) -> fn x xs
    )
last :: [a] -> a
last xs = case xs of
  [] -> error "last: []"
  (x:xs) -> work x xs

work :: a -> [a] -> a
work = fix ( \ fn x xs ->
  case xs of
    [] -> x
    (x:xs) -> fn x xs
  )
last :: [a] -> a
last xs = case xs of
  [] -> error "last: []"
  (x:xs) -> work x xs

work :: a -> [a] -> a
work x xs = case xs of
  [] -> x
  (x:xs) -> work x xs
When does the Worker/Wrapper Transformation Succeed?

When $\text{unwrap} \cdot \text{wrap}$ fuse!

Emerging heuristic...

Simplification Friendly

Pre-conditions:
- any of basic, body or fix
- $\text{unwrap} \cdot \text{wrap} = \text{id}_B$

When $A$ is "larger" than $B$

Worker/Wrapper Fusion

Pre-condition:
- $\text{wrap} \cdot \text{unwrap} = \text{id}_A$

When $B$ is "larger" than $A$

More powerful fusion methods can also be used.
Creating Workers and Wrappers for reverse

reverse :: [a] → [a]

unwrap :: [a] → H a

work :: [a] → H a

wrap :: H a → [a]

unwrap fn = \ xs → c2a (fn xs)

wrap fn = \ xs → a2c (fn xs)

type H a = [a] → [a]

a2c :: H a → [a]
a2c f = f []
c2a :: [a] → H a
c2a xs = \ ys → xs ++ ys
Testing the basic worker/wrapper assumption: Does \( \text{wrap} \cdot \text{unwrap} = \text{id} \)?

\[
\text{wrap} \cdot \text{unwrap} = \{ \text{apply wrap, unwrap} \} \\
(\lambda \text{fn xs} \to \text{a2c (fn xs)}) \cdot (\lambda \text{fn xs} \to \text{c2a (fn xs)}) \\
= \{ \text{apply } \cdot \} \\
\lambda \text{f} \to (\lambda \text{fn xs} \to \text{a2c (fn xs)}) ((\lambda \text{fn xs} \to \text{c2a (fn xs)}) \text{f}) \\
= \{ \beta\text{-reduction} \} \\
\lambda \text{f} \to (\lambda \text{fn xs} \to \text{a2c (fn xs)}) (\lambda \text{xs} \to \text{c2a (f xs)}) \\
= \{ \beta\text{-reduction} \} \\
\lambda \text{f} \to (\lambda \text{xs} \to \text{a2c ((\lambda \text{xs} \to \text{c2a (f xs)}) \text{xs})}) \\
= \{ \beta\text{-reduction} \} \\
\lambda \text{f} \to (\lambda \text{xs} \to \text{a2c (c2a (f xs)))}
\]
Does $a_2c \cdot c_2a = \text{id}$?

\[
\begin{align*}
a_2c \cdot c_2a &= \{ \text{apply } a_2c, \ c_2a \} \\
(\lambda f \to f \; []) \cdot (\lambda \; xs \; ys \to \; xs \; ++ \; ys) &= \{ \text{apply } \cdot \} \\
\lambda \; zs \to (\lambda f \to f \; []) \; ((\lambda \; xs \; ys \to \; xs \; ++ \; ys) \; zs) &= \{ \beta\text{-reduction} \} \\
\lambda \; zs \to (\lambda f \to f \; []) \; (\lambda \; ys \to \; zs \; ++ \; ys) &= \{ \beta\text{-reduction} \} \\
\lambda \; zs \to (\lambda \; ys \to \; zs \; ++ \; ys) \; [] &= \{ \beta\text{-reduction} \} \\
\lambda \; zs \to \; zs \; ++ \; [] &= \{ \text{[]} \text{ is the identity for } ++ \} \\
\lambda \; zs \to \; zs
\end{align*}
\]
Improving Reverse

\[
\text{reverse} :: [a] \rightarrow [a] \\
\text{work} :: [a] \rightarrow H\ a
\]

\[
\text{wrap} \quad \text{fn} = \lambda \ x s \rightarrow \text{a2c} \ (\text{fn} \ x s)
\]
\[
\text{unwrap} \quad \text{fn} = \lambda \ x s \rightarrow \text{c2a} \ (\text{fn} \ x s)
\]

\[
\text{body} \quad \text{rev} = \lambda \ v \rightarrow \text{case} \ v \text{ of}
\]
\[
[\ ] \quad \rightarrow \ [\ ]
\]
\[
(x:xs) \quad \rightarrow \ \text{rev} \ x s \ +\ + \ [x]
\]

\[
\text{reverse} = \text{wrap} \ \text{work}
\]

\[
\text{work} = \text{fix} \ (\text{unwrap} \ . \ \text{body} \ . \ \text{wrap})
\]
Improving Reverse (2)

reverse = (\ fn xs -> a2c (fn xs)) work

work = fix ( (\ fn xs -> c2a (fn xs))
  . (\ rev v -> case v of
      [] -> []
      (x:xs) -> rev xs ++ [x])
  . (\ fn xs -> a2c (fn xs))
)

深加工 · 使得简化使用 \(\beta\)-归纳法。
reverse xs = \a2c (work xs)

work = fix (\rev v -> c2a (case v of
          [] -> []
          (x:xs) -> a2c (rev xs) ++ [x]))

- We now have the structure to being something akin to rippling.
- Goal:
  - Move the c2a to just in front of a2c, giving \boxed{\text{c2a (a2c (rev xs))}}
  - Unapply unwrap and unwrap, giving \boxed{\text{unwrap (wrap rev)}}
  - Use the Worker/Wrapper fusion law

**The Worker/Wrapper Fusion Property**

If \(\text{wrap} \cdot \text{unwrap} = \text{id}\), then \((\text{unwrap} \cdot \text{wrap}) \text{work} = \text{work}\)
reverse $xs = a2c \ (work \ xs)$

\[
\begin{aligned}
work &= \text{fix} (\ \lambda \ rev \ v \rightarrow c2a \ (\text{case} \ v \ \text{of} \\
&\hspace{1cm} [] \rightarrow [] \\
&\hspace{1cm} (x:xs) \rightarrow a2c \ (\text{rev} \ xs) ++ [x]) \\
\end{aligned}
\]

- We use distribution over case to push $c2a$ inside the case expression
- This relies on $c2a$ being strict
reverse \(xs\) = \(a2c\) (work \(xs\))

\[
\begin{align*}
\text{work} &= \text{fix} (\lambda \text{rev v} \rightarrow \text{case v of} \\
&\quad \quad [] \rightarrow c2a [] \\
&\quad (x:xs) \rightarrow c2a (a2c (\text{rev xs}) ++ [x])) \\
&\end{align*}
\]

- We use distribution over case to push \(c2a\) inside the case expression
- This relies on \(c2a\) being strict
- \(c2a\) has made progress towards our goal
reverse $xs = a2c \ (work \ xs)$

$$work = fix (\ rev \ v \rightarrow \ case \ v \ of \ [ ] \rightarrow id \ \ (x:xs) \rightarrow c2a \ (a2c \ (rev \ xs) ++ [x]))$$

- We use distribution over case to push $c2a$ inside the case expression
- This relies on $c2a$ being strict
- $c2a$ has made progress towards our goal
- $c2a \ [] = id$ (by applying $c2a$, ++)
reverse \( xs = a2c \left( \text{work} \ xs \right) \)

\[
\text{work} = \text{fix} \left( \lambda \text{rev} \ v \rightarrow \text{case} \ v \ \text{of} \right) \\
\quad \left[ \right] \rightarrow \text{id} \\
\quad \left( x : xs \right) \rightarrow \ c2a \ \left( a2c \ \right. \left. \left( \text{rev} \ xs \right) ++ \left[ x \right] \right) \\
\)

\[
\bullet \ c2a \ \left( e1 \ ++ \ e2 \right) = c2a \ e1 \ \cdot \ c2a \ e2
\]
reverse \( xs = a2c \ (work \ xs) \)

\[
work = fix (\ rev \ v \to \ case \ v \ of \\
    [] \to id \\
    (x:xs) \to c2a \ (a2c \ (rev \ xs)) \cdot c2a \ [x]
\)

- \( c2a \ (e1 \ ++ \ e2) = c2a \ e1 \cdot c2a \ e2 \)
- We can now unapply \( \text{unwrap} \) and \( \text{wrap} \)
reverse xs = a2c (work xs)

work = fix (\ rev v -> case v of
               []    -> id
               (x:xs) -> (unwrap . wrap) rev xs . c2a [x]
           )

- c2a (e1 ++ e2) = c2a e1 \cdot c2a e2
- We can now unapply unwrap and wrap
- And we can use the Worker/Wrapper Fusion Property
Improving Reverse (5)

\[ \text{reverse } xs = \text{a2c} \left( \text{work } xs \right) \]

\[ \text{work} = \text{fix} \left( \lambda \text{rev } v \rightarrow \text{case } v \text{ of} \right) \]
\[ \quad \quad \quad \left[ \right] \rightarrow \text{id} \]
\[ \quad \quad \quad \left( x : xs \right) \rightarrow \text{rev } xs \cdot \text{c2a } [x] \]
\}

- \[ \text{c2a } (e1 + e2) = \text{c2a } e1 \cdot \text{c2a } e2 \]
- We can now unapply un\text{wrap} and wrap
- And we can use the \text{Worker/Wrapper Fusion Property}
- We have removed the overhead of the coercion
reverse xs = work xs []

work = fix (\ rev v -> case v of
            []     -> \ ys -> ys
            (x:xs) -> \ ys -> rev xs (x : ys)
)

- After further uses of apply, we reach a clean implementation using fix
- We now can apply fix (and other small transformations)
reverse xs = work xs []

work [] ys = ys
work (x:xs) ys = work xs (x:ys)

- After further uses of apply, we reach a clean implementation using \texttt{fix}
- We now can apply \texttt{fix} (and other small transformations)
- Efficient reverse!
Conclusions

- Worker/wrapper is a general and systematic approach to transforming a computation of one type into an equivalent computation of another type.
- It is straightforward to understand and apply, requiring only basic equational reasoning techniques, and often avoiding the need for induction.
- It allows many seemingly unrelated optimization techniques to be captured inside a single unified framework.
Further Work

- Monadic and Effectful Constructions
- Mechanization
- Implement inside the Haskell Equational Reasoning Assistant
- Consider other patterns of recursion

www.workerwrapper.com