The Worker/Wrapper Transformation

Andy Gill[1] Graham Hutton[2]

[1]Galois, Inc.

[2]University of Nottingham

March 11, 2008

The Worker/Wrapper Transformation

> 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

```
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 *Int#*, a strict version of Int that is passed by value for `n` and `m`

After

```
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

```
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

```
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

```
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

```
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

```
eval :: Expr -> Maybe Int
eval (Val n)      =  Just n
eval (Add x y)    =  case eval x of Nothing -> Nothing
                                    Just n  -> case eval y of Nothing -> Nothing
                                                              Just m  -> Just (n+m)
eval (Throw)      =  Nothing
eval (Catch x y) =  case eval x of Nothing -> eval y
                                    Just n  -> Just n
```

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

After

```
eval e = work e Just Nothing

work :: Expr -> (Int -> Maybe Int) -> Maybe Int -> Maybe Int
work (Val n)     s f  =  s n
work (Add x y)   s f  =  work x (\n -> work y (\m -> s (n+m)) f) f
work (Throw)     s f  =  f
work (Catch x y) s f  =  work x s (work y s f)
```

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

```
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 last

```
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
- These functions are mutually recursive

Inline Wrapper

```
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`

Inline Wrapper

```
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 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

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 ->
                              case xs of
                                []     -> x
                                (x:xs) -> last_work x xs
```

- We now simplify the worker
- Reaching our efficient implementation

The Informal Worker Wrapper Methodology

- From a recursive function, construct two new functions

Wrapper

- Replacing the original function
- Coerces call to Worker

Worker

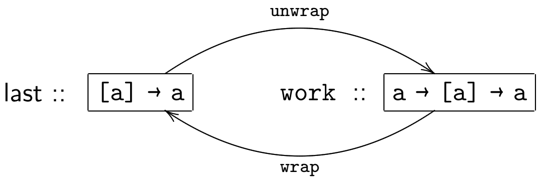
- Performs main computation
- Syntactically contains the body of the original function
- Coerces call from Wrapper

- 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

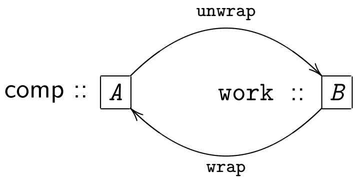
Questions about the Worker/Wrapper Transformation

- 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



wrap and unwrap in General



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

$\texttt{wrap} \cdot \texttt{unwrap} = \texttt{id}_A$ *(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

$\texttt{wrap} \cdot \texttt{unwrap} = \texttt{id}_A$ *(basic worker/wrapper assumption)*

Derivation

`comp = fix body`
$=$ { id is the identity for $\cdot$ }
`comp = fix (id · body)`

Prerequisites

`comp` :: A
`comp = fix body` for some `body` :: $A \to A$

`wrap` :: $B \to A$ is a coercion from type B to A
`unwrap` :: $A \to B$ is a coercion from type A to B

`wrap` $\cdot$ `unwrap` $= \mathtt{id}_A$ *(basic worker/wrapper assumption)*

Derivation

`comp = fix body`
$=$ { `id` is the identity for $\cdot$ }
`comp = fix (id · body)`
$=$ { assuming `wrap · unwrap = id` }
`comp = fix (wrap · unwrap · 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

$\texttt{wrap} \cdot \texttt{unwrap} = \texttt{id}_A$ *(basic worker/wrapper assumption)*

Derivation

`comp = fix body`
= { id is the identity for · }
`comp = fix (id · body)`
= { assuming `wrap · unwrap = id` }
`comp = fix (wrap · unwrap · body)`
= { rolling rule }
`comp = wrap (fix (unwrap · body · wrap))`

Prerequisites

comp :: A

comp = fix body for some body :: $A \to A$

wrap :: $B \to A$ is a coercion from type B to A

unwrap :: $A \to B$ is a coercion from type A to B

wrap $\cdot$ unwrap $=$ id_A *(basic worker/wrapper assumption)*

Derivation

comp = fix body

$=$ { id is the identity for $\cdot$ }

comp = fix (id $\cdot$ body)

$=$ { assuming wrap $\cdot$ unwrap = id }

comp = fix (wrap $\cdot$ unwrap $\cdot$ body)

$=$ { rolling rule }

comp = wrap (fix (unwrap $\cdot$ body $\cdot$ wrap))

$=$ { define work = fix (unwrap $\cdot$ body $\cdot$ wrap) }

comp = wrap work

work = fix (unwrap $\cdot$ body $\cdot$ 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

$\texttt{wrap} \cdot \texttt{unwrap} = \texttt{id}_A$ *(basic worker/wrapper assumption)*

Worker/Wrapper Theorem

If the above prerequisites hold, then

```
comp = fix body
```

can be rewritten as

```
comp =  wrap work
```

where `work :: B` is defined by

```
work =  fix (unwrap · body · wrap)
```

The Worker/Wrapper Assumptions

Key step of proof

$\texttt{fix (id} \cdot \texttt{body)}$

$= \{ \text{assuming } \texttt{wrap} \cdot \texttt{unwrap = id} \}$

$\texttt{fix (wrap} \cdot \texttt{unwrap} \cdot \texttt{body)}$

We can actually use any of three different assumptions here

$$\begin{array}{rclr}
\texttt{wrap} \cdot \texttt{unwrap} & = & \texttt{id} & \text{(basic assumption)} \\
& \Downarrow & & \\
\texttt{wrap} \cdot \texttt{unwrap} \cdot \texttt{body} & = & \texttt{body} & \text{(body assumption)} \\
& \Downarrow & & \\
\texttt{fix (wrap} \cdot \texttt{unwrap} \cdot \texttt{body)} & = & \texttt{fix body} & \text{(fix-point assumption)}
\end{array}$$

The Worker/Wrapper Recipe

Recipe

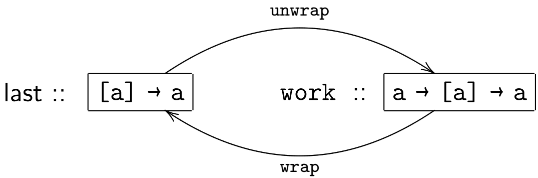
- Express the computation as a least fixed point;
- Choose the desired new type for the computation;
- Define conversions between the original and new types;
- Check they satisfy one of the worker/wrapper assumptions;
- Apply the worker/wrapper transformation;
- Simplify the resulting definitions.

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

If $\mathtt{wrap} \cdot \mathtt{unwrap} = \mathtt{id}$, then $(\mathtt{unwrap} \cdot \mathtt{wrap})\ \mathtt{work} = \mathtt{work}$

Creating Workers and Wrappers for last




```
wrap fn = \ xs -> case xs of
                    [] -> error "last: []"
                    (x:xs) -> fn x xs

unwrap fn = \ x xs -> fn (x:xs)

last = fix body

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

Testing the basic worker/wrapper assumption: Does wrap · unwrap = id?

```
wrap . unwrap
```

= { apply wrap, unwrap and · }

```
\ fn ->
  (\ xs -> case xs of
             [] -> error "last: []"
             (x:xs) -> (\ x xs -> fn (x:xs)) x xs)
```

= { β-reduction }

```
\ fn ->
  (\ xs -> case xs of
             [] -> error "last: []"
             (x:xs) -> fn (x:xs))
```

Clearly not equal to `id :: ([a] → a) → ([a] → a)`

Testing the body worker/wrapper assumption: Does wrap · unwrap · body = body?

```
wrap . unwrap . body
```

= { apply `wrap`, `unwrap` and · }

```
(\ fn ->
  (\ xs -> case xs of
             [] -> error "last: []"
             (x:xs) -> (\ x xs -> fn (x:xs)) x xs))
   (\ last v -> case v of
                  []     -> error "last: []"
                  (x:[]) -> x
                  (x:xs) -> last xs)
```

Testing the body worker/wrapper assumption: Does `wrap` · `unwrap` · `body` = `body`?

```
wrap . unwrap . body
```

= { apply `wrap`, `unwrap` and · }
= { β-reductions }

```
(\ fn ->
  (\ xs -> case xs of
             [] -> error "last: []"
             (x:xs) -> case (x:xs) of
                         []     -> error "last: []"
                         (x:[]) -> x
                         (x:xs) -> fn xs))
```

Testing the body worker/wrapper assumption: Does `wrap` · `unwrap` · `body` = `body`?

```
wrap . unwrap . body
```

= { apply `wrap`, `unwrap` and · }
= { β-reductions }
= { case of known constructors }

```
(\ fn ->
  (\ xs -> case xs of
             [] -> error "last: []"
             (x:xs) -> case xs of
                         [] -> x
                         xs -> fn xs))
```

Testing the body worker/wrapper assumption: Does `wrap` · `unwrap` · `body` = `body`?

```
wrap . unwrap . body
```

= { apply `wrap`, `unwrap` and · }
= { β-reductions }
= { case of known constructors }
= { common up case }

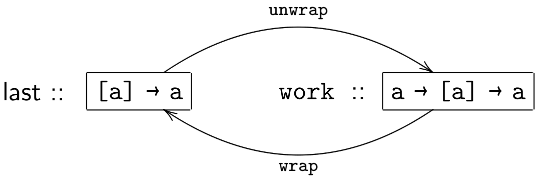
```
(\ fn ->
  (\ xs -> case xs of
              [] -> error "last: []"
              (x:[]) -> x
              (x:xs) -> fn xs))
```

Which equals `body`. QED.

Applying the Worker/Wrapper Transformation

Before

```
last = fix body
```




```
last  :: [a] -> a
last = wrap work

work :: a -> [a] -> a
work = fix ( unwrap
           . body
           . wrap
           )
```

Inline and Simplify

```
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)
           )
```

Inline and Simplify

```
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
          )
```

Inline and Simplify

```
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
          )
```

Inline and Simplify

```
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
          )
```

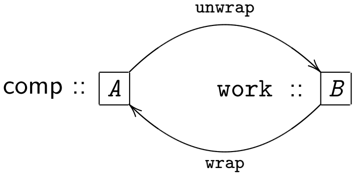
Inline and Simplify

```
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 `unwrap` · `wrap` fuse!



Emerging heuristic...

Simplification Friendly

Pre-conditions:

any of basic, body or fix

`unwrap · wrap = id`$_B$

When A is "larger" than B

Worker/Wrapper Fusion

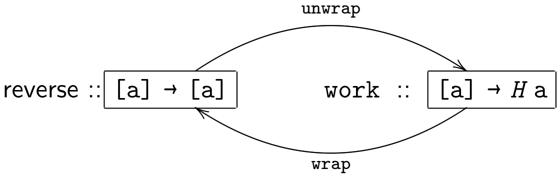
Pre-condition:

`wrap · unwrap = id`$_A$

When B is "larger" than A

More powerful fusion methods can also be used

Creating Workers and Wrappers for reverse




```
type H a = [a] -> [a]

a2c :: H a -> [a]
a2c f = f []

c2a :: [a] -> H a
c2a xs = \ ys -> xs ++ ys

wrap fn = \ xs -> a2c (fn xs)
unwrap fn = \ xs -> c2a (fn xs)
```

Testing the basic worker/wrapper assumption: Does `wrap` · `unwrap` = `id`?

```
wrap . unwrap
```

= { apply `wrap`, `unwrap` }

```
(\ fn xs -> a2c (fn xs)) . (\ fn xs -> c2a (fn xs))
```

= { apply · }

```
\ f -> (\ fn xs -> a2c (fn xs)) ((\ fn xs -> c2a (fn xs)) f)
```

= { β-reduction }

```
\ f -> (\ fn xs -> a2c (fn xs)) (\ xs -> c2a (f xs))
```

= { β-reduction }

```
\ f -> (\ xs -> a2c ((\ xs -> c2a (f xs)) xs))
```

= { β-reduction }

```
\ f -> (\ xs -> a2c (c2a (f xs)))
```

Does a2c · c2a = id?

```
a2c . c2a
```

= { apply a2c, c2a }

```
(\ f -> f []) . (\ xs ys -> xs ++ ys)
```

= { apply · }

```
\ zs -> (\ f -> f []) ((\ xs ys -> xs ++ ys) zs)
```

= { β-reduction }

```
\ zs -> (\ f -> f []) (\ ys -> zs ++ ys)
```

= { β-reduction }

```
\ zs -> (\ ys -> zs ++ ys) []
```

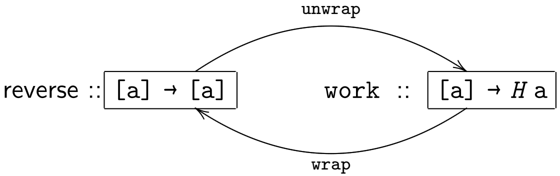
= { β-reduction }

```
\ zs -> zs ++ []
```

= { [] is the identity for ++ }

```
\ zs -> zs
```

Improving Reverse




```
wrap fn = \ xs -> a2c (fn xs)
unwrap fn = \ xs -> c2a (fn xs)

body rev = \ v -> case v of
                    []     -> []
                    (x:xs) -> rev xs ++ [x]

reverse = wrap work

work = fix ( unwrap . body . 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))
           )
```

- Inline $\cdot$ and simplify using β-reduction.

Improving Reverse (3)

```
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 c2a (a2c (rev xs))
  - Unapply `unwrap` and `unwrap`, giving unwrap (wrap rev)
  - Use the Worker/Wrapper fusion law

The Worker/Wrapper Fusion Property

If `wrap` · `unwrap` = `id`, then (`unwrap` · `wrap`) `work` = `work`

Improving Reverse (4)

```
reverse xs = a2c (work xs)

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

- We use distribution over case to push c2a inside the case expression
- This relies on c2a being strict

Improving Reverse (4)

```
reverse xs = a2c (work xs)

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

Improving Reverse (4)

```
reverse xs = a2c (work xs)

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

Improving Reverse (5)

```
reverse xs = a2c (work xs)

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

- c2a (e1 ++ e2) = c2a e1 · c2a e2

Improving Reverse (5)

```
reverse xs = a2c (work xs)

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

- c2a (e1 ++ e2) = c2a e1 · c2a e2
- We can now unapply `unwrap` and `wrap`

Improving Reverse (5)

```
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 · c2a e2
- We can now unapply `unwrap` and `wrap`
- And we can use the Worker/Wrapper Fusion Property

Improving Reverse (5)

```
reverse xs = a2c (work xs)

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

- c2a (e1 ++ e2) = c2a e1 · c2a e2
- We can now unapply `unwrap` and `wrap`
- And we can use the Worker/Wrapper Fusion Property
- We have removed the overhead of the coercion

Improving Reverse (5)

```
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)

Improving Reverse (5)

```
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 `fix`
- We now can apply `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