

# The Worker/Wrapper Transformation

Andy Gill<sup>1</sup>   Graham Hutton<sup>2</sup>

<sup>1</sup>Galois, Inc.

<sup>2</sup>University of Nottingham

February 12, 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 than 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 a implementation

Thesis:

- The Worker/Wrapper Transformation is 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  $(A \rightarrow X) \rightarrow X$
- 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 a 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

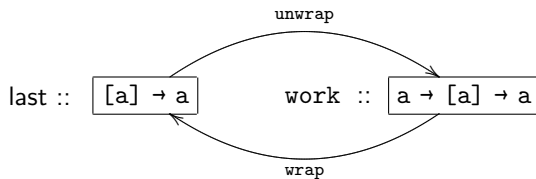
- Performs main computation
- Syntactically contains the body of the original function
- Coerces call from Wrapper

- 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

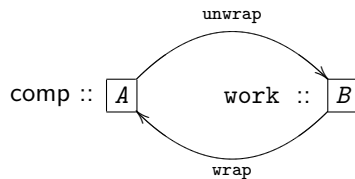
# Questions about the Worker/Wrapper Transformation

- 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

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  to  $B$

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

## Derivation

$\text{comp} = \text{fix body}$

## Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  to  $B$

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

## Derivation

$\text{comp} = \text{fix body}$

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

$\text{comp} = \text{fix} (\text{id} \cdot \text{body})$

## Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  to  $B$

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

## Derivation

$\text{comp} = \text{fix body}$

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

$\text{comp} = \text{fix} (\text{id} \cdot \text{body})$

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

**$\text{comp} = \text{fix} (\text{wrap} \cdot \text{unwrap} \cdot \text{body})$**

## Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  to  $B$

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

## Derivation

$\text{comp} = \text{fix body}$

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

$\text{comp} = \text{fix} (\text{id} \cdot \text{body})$

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

$\text{comp} = \text{fix} (\text{wrap} \cdot \text{unwrap} \cdot \text{body})$

$= \{ \text{rolling rule} \}$

$\text{comp} = \text{wrap} (\text{fix} (\text{unwrap} \cdot \text{body} \cdot \text{wrap}))$

## Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  to  $B$

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

## Derivation

$\text{comp} = \text{fix body}$

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

$\text{comp} = \text{fix} (\text{id} \cdot \text{body})$

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

$\text{comp} = \text{fix} (\text{wrap} \cdot \text{unwrap} \cdot \text{body})$

$= \{ \text{rolling rule} \}$

$\text{comp} = \text{wrap} (\text{fix} (\text{unwrap} \cdot \text{body} \cdot \text{wrap}))$

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

$\text{comp} = \text{wrap work}$

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

## Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$  is a coercion from type  $B$  to  $A$

$\text{unwrap} :: A \rightarrow B$  is a coercion from type  $A$  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

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

# The Worker/Wrapper Assumptions

## Key step of proof

```
fix (id · body)
= { assuming wrap · unwrap = id }
fix (wrap · unwrap · body)
```

We can actually use any of three different assumptions here

$\text{wrap} \cdot \text{unwrap}$	$=$	$\text{id}$	(basic assumption)
	$\Downarrow$		
$\text{wrap} \cdot \text{unwrap} \cdot \text{body}$	$=$	$\text{body}$	(body assumption)
	$\Downarrow$		
$\text{fix} (\text{wrap} \cdot \text{unwrap} \cdot \text{body})$	$=$	$\text{fix body}$	(fix-point assumption)

# The Worker/Wrapper Recipe

## Recipe

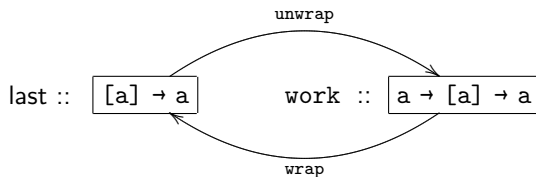
- Express the computation as a least fixed point;
- Choose the desired new type for the computation;
- Define conversions between the original and new types;
- Check they satisfy one of the worker/wrapper assumptions;
- Apply the worker/wrapper transformation;
- Simplify the resulting definitions.

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

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

# 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 $\text{wrap} \cdot \text{unwrap} = \text{id}$ ?

$\text{wrap} \cdot \text{unwrap}$

$= \{ \text{apply wrap, unwrap and } \cdot \}$

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

$= \{ \beta\text{-reduction} \}$

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

Clearly not equal to  $\text{id} :: ([a] \rightarrow a) \rightarrow ([a] \rightarrow a)$

# Testing the body worker/wrapper assumption: Does $\text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body}$ ?

```
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 $\text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body}$ ?

```
wrap . unwrap . body
```

```
= { apply wrap, unwrap and . }
```

```
= {  $\beta$ -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 $\text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body}$ ?

```
wrap . unwrap . body
```

```
= { apply wrap, unwrap and . }
```

```
= {  $\beta$ -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 $\text{wrap} \cdot \text{unwrap} \cdot \text{body} = \text{body}$ ?

`wrap . unwrap . body`

= { apply wrap, unwrap and  $\cdot$  }  
= {  $\beta$ -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

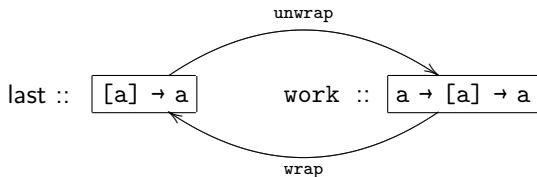
```
last = fix body
```

```
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
    )
```

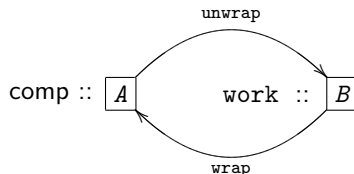
# Inline and Simplify

```
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  $\text{unwrap} \cdot \text{wrap}$  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

# 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