

The Worker/Wrapper Transformation

Andy Gill¹ Graham Hutton²

¹Galois, Inc.

²University of Nottingham

February 6, 2008

The Worker/Wrapper Transformation?

What is the Worker/Wrapper Transformation?

The worker wrapper transformation is a rewriting technique which changes the type of a recursive computation

Changing the type of a computation ...

- is pervasive in functional programming
 - useful in practice
 - the essence of turning a specification into a implementation
-
- Worker/wrapper has been used inside the Glasgow Haskell compiler since its inception to rewriting functions that take lifted values (thunks) with unlifted, equivalent values.
 - No direct proof of correctness; syntactical arguments given

This Talk

- Give example of traditional use of the Worker/Wrapper Transformation
 - Deriving efficient `last`
- Formalize the Worker/Wrapper Transformation
 - Using the rolling rule and basic functional programming
- Apply worker/wrapper recipe to some examples
 - Revisit efficient `last`
 - Deriving memoization
 - Deriving the CPS translation
- Draw conclusions about general applicability

Example: Fetching the last element in a list in Haskell

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

Example: Fetching the last element in a list in Haskell

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

- If we desugar into “Core” Haskell, `last` looks like this

```
last = λ v → case v of  
      []      → error "last: []"  
      (x:xs) → case xs of  
              []      → x  
              (_:_) → last xs
```

- There are two **case** statements per recursive call
- Any recursive call of `last` will always contain at least one cons cell

Workers and Wrappers for last

A possible alternative definition of last might be

```
last :: [a] → a
last = λ v → case v of
    []      → error "last: []"
    (x:xs) → last_work x xs

last_work :: a → [a] → a
last_work = λ x xs → case xs of
    []      → x
    (x':xs') → last_work x' xs'
```

- We have a *wrapper* last which acts as an impedance match between last and last_work.
- We have a *worker* last_work which does the recursive computation
- There now is only one **case** statement per recursive call
- This talk: Systematically deriving workers and wrappers like these.

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.

β -reduce worker

```
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 perform β -reduction

β -reduce worker

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

- We now perform β -reduction

β -reduce worker

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

- We now perform β -reduction

β -reduce worker

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

- We now perform β -reduction

Case rules

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

- We have two instances of case on the same value

Case rules

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

- We have two instances of case on the same value
- We can merge these case, removing redundant branches

Case of known constructor

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

- We have case of a known constructor, cons.

Case of known constructor

```
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 have case of a **known** constructor, cons.
- **Being careful about naming, can remove the case**
- We have reached our efficient implementation

Case of known constructor

```
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 have case of a **known** constructor, cons.
- Being careful about naming, can remove the case
- **We have reached 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

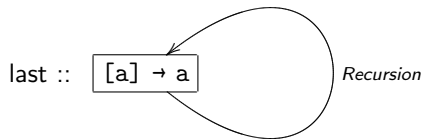
- Performs main computation
- Syntactically contains the body of the original function
- Coerces call from Wrapper

- 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

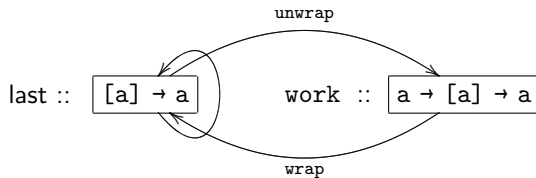
Questions about the Worker/Wrapper Transformation

- Is the technique actually correct?
- How can this be proved?
- Under what conditions does it hold?
- How should it be used in practice?
- What kind of applications is it suitable for?

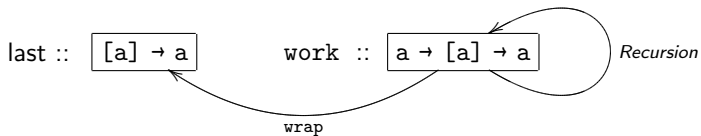
wrap and unwrap



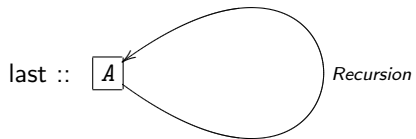
wrap and unwrap



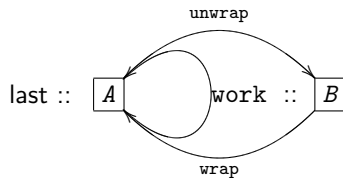
wrap and unwrap



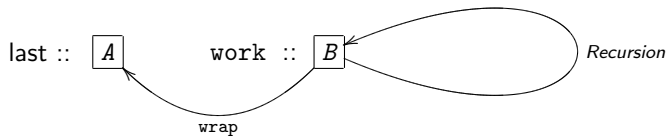
wrap and unwrap in General



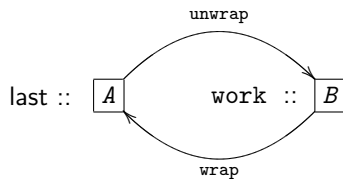
wrap and unwrap in General



wrap and unwrap in General



Key wrap and unwrap Diagram



Prerequisites

$\text{comp} :: A$

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

$\text{wrap} :: B \rightarrow A$ 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

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

can be rewritten as

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

where $\text{work} :: B$ is defined by

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

The rolling rule

$$\text{fix}(f.g) = f(\text{fix}(g.f))$$

Intuitively correct because both sides expand into

$$f(g(f(g(f(g(\dots)))))))$$

Proof of Worker/Wrapper Theorem

```
comp
= { applying comp }

fix body
= { id is the identity for · }

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

fix (wrap · unwrap · body)
= { rolling rule }

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

wrap work
```

The worker/wrapper assumptions

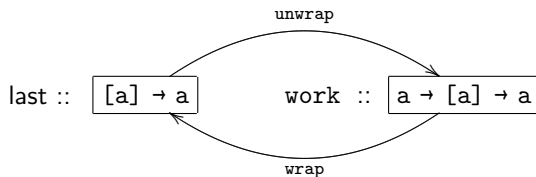
Key step of proof

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

We can actually use any of three different assumptions here

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

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

`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

```
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

```
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

`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

`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

```
last  :: [a] -> a
```

```
last = wrap work
```

```
work  :: a -> [a] -> a
```

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

Simplifying work

```
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)
    )
```

Simplifying work

```
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
    )
```

Simplifying work

```
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
    )
```

Simplifying work

```
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
    )
```

Simplifying work

```
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
```

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 worker/wrapper fusion property.

The Worker/Wrapper Fusion Property

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

Deriving Memoisation

```
fib :: Int → Int
fib n =
  if n < 2
  then 1
  else fib (n-1) + fib (n-2)
```

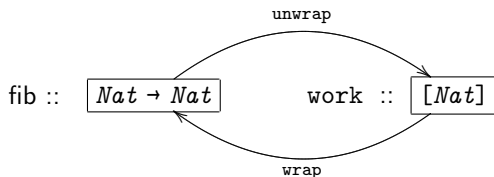
```
wrap :: [Nat] → (Nat → Nat)
wrap xs ix = xs !! ix
```

```
unwrap :: (Nat → Nat) → [Nat]
unwrap f = map f [0..]
```

After proving that $\text{wrap} \cdot \text{unwrap} = \text{id}$, and applying rewrites, we reach

```
fib :: Nat → Nat
fib n = work !! n
```

```
work = map f [0..]
  where f = if n < 2 then 1 else work !! (n-1) + work !! (n-2)
```



Deriving a CPS version of a small evaluator

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

```
eval      :: Expr → Maybe Int
```

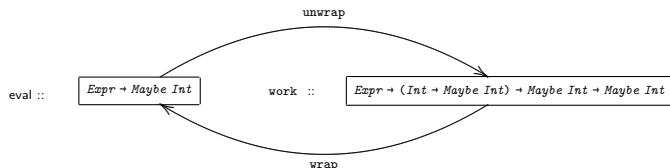
```
eval (Val n) = Just n
```

```
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 wrap and unwrap for our small evaluator



```
unwrap :: (Expr → Maybe Int)
        → (Expr → (Int → Maybe Int) → Maybe Int → Maybe Int)
unwrap g e s f = case (g e) of
                  Nothing → f
                  Just n  → s n
```

```
wrap :: (Expr → (Int → Maybe Int) → Maybe Int → Maybe Int)
       → (Expr → Maybe Int)
wrap g e = g e Just Nothing
```

Result of deriving a CPS version of a small evaluator

```
work                                :: Expr
                                   → (Int → Maybe Int)
                                   → Maybe Int
                                   → Maybe Int

work (Val n)      s f = s n
work (Add x y)    s f = work x (λn → work y (λm → s (n+m))) f) f
work (Throw)      s f = f
work (Catch x y) s f = work x s (work y s f)
```

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