The Worker/Wrapper Transformation

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

\textsuperscript{1}Galois, Inc.

\textsuperscript{2}University of Nottingham

February 12, 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 unplifted 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

```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

\[
\begin{align*}
\text{last} & \quad :: \quad [a] \rightarrow a \\
\text{last} [\ ] & = \text{error} \ "last: \ []" \\
\text{last} \ (x:[]) & = x \\
\text{last} \ (x:xs) & = \text{last} \ xs
\end{align*}
\]

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

After

\[
\begin{align*}
\text{last} [\ ] & = \text{error} \ "last: \ []" \\
\text{last} \ (x:xs) & = \text{work} \ x \ xs
\end{align*}
\]

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

Andy Gill, Graham Hutton
Example 3: Efficient \texttt{nub}

\textbf{Before}

\begin{verbatim}
 nub :: [Int] -> [Int]
nub [] = []
nub (x:xs) = x : nub (filter (\y -> not (x == y)) xs)
\end{verbatim}

- \texttt{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.

\textbf{After}

\begin{verbatim}
 nub :: [Int] -> [Int]
nub xs = work xs empty

work :: [Int] -> Set Int -> [Int]
work xs except =
    case dropWhile (\ x -> x \textquoteleft member\textquoteright{} except) xs of
        [] -> []
        (x:xs) -> x : work xs (insert x except)
\end{verbatim}
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

```haskell
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

```haskell
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

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

```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
- These functions are mutually recursive
last :: [a] -> a
last = \ v -> case v of
    []  -> error "last: []"
    (x:xs) -> last_work x xs

case xs of
    []  -> x
    (_:_)) -> last xs) (x:xs)

We now inline last inside last_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 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

\[
\begin{align*}
\text{last} &:: [a] \to a \\
\text{work} &:: a \to [a] \to a
\end{align*}
\]
wrap and unwrap in General

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

unwrap

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

\[ \text{comp} = \text{fix body} \]
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  \quad (basic worker/wrapper assumption)

Derivation

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

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

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

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

Derivation

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

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

Derivation

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

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

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

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

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

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

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

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

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

\[
\text{fix (id ⋅ body)} = \{ \text{assuming wrap ⋅ unwrap = id} \} \text{fix (wrap ⋅ unwrap ⋅ body)}
\]

We can actually use any of three different assumptions here

\[
\begin{align*}
\text{wrap ⋅ unwrap} & = \text{id} \quad \text{(basic assumption)} \\
\downarrow & \\
\text{wrap ⋅ unwrap ⋅ body} & = \text{body} \quad \text{(body assumption)} \\
\downarrow & \\
\text{fix (wrap ⋅ unwrap ⋅ body)} & = \text{fix body} \quad \text{(fix-point assumption)}
\end{align*}
\]
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 \texttt{wrap} and \texttt{unwrap} coercions, often using fusion, including the worker/wrapper fusion property.

The Worker/Wrapper Fusion Property

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

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

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

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

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

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

\[
\text{wrap} \cdot \text{unwrap} = \{ \text{apply wrap, unwrap and } \cdot \} \\
\text{\textbackslash fn } \rightarrow \text{\textbackslash xs } \rightarrow \text{case xs of} \\
\hspace{1cm} [] \rightarrow \text{error "last: []"} \\
\hspace{1cm} (x:xs) \rightarrow (\text{\textbackslash x xs } \rightarrow \text{fn (x:xs)}) x xs
\]

\[
= \{ \beta\text{-reduction} \} \\
\text{\textbackslash fn } \rightarrow \text{\textbackslash xs } \rightarrow \text{case xs of} \\
\hspace{1cm} [] \rightarrow \text{error "last: []"} \\
\hspace{1cm} (x:xs) \rightarrow \text{fn (x:xs)}
\]

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

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

\[
\begin{align*}
\text{wrap} \cdot \text{unwrap} \cdot \text{body} &= \{ \text{apply wrap, unwrap and } \cdot \} \\
&= \{ \beta\text{-reductions} \} \\
&= (\lambda \text{fn} -> (\lambda \text{xs} -> \text{case xs of}
\begin{align*}
[] &\rightarrow \text{error "last: []"} \\
(x:xs) &\rightarrow \text{case (x:xs) of}
\begin{align*}
[] &\rightarrow \text{error "last: []"} \\
(x:[]) &\rightarrow x \\
(x:xs) &\rightarrow \text{fn xs})
\end{align*}
)\).
\end{align*}
\]
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} \} \\

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

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

\[
(\ \lambda \text{fn} \to \ \\
(\ \lambda \text{xs} \to \text{case xs of} \ \\
\hspace{1em}[] \to \text{error "last: []"} \ \\
\hspace{1em}(\text{x:[]}) \to \text{x} \ \\
\hspace{1em}(\text{x:xs}) \to \text{fn xs})

Which equals body. QED.
Applying the Worker/Wrapper Transformation

Before

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

\[
\begin{align*}
\text{last} :: & \quad [a] \rightarrow a \\
\text{last} = & \quad \text{wrap work}
\end{align*}
\]

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