The Worker/Wrapper Transformation

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

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

February 6, 2008
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 \texttt{last}
- Formalize the Worker/Wrapper Transformation
  - Using the rolling rule and basic functional programming
- Apply worker/wrapper recipe to some examples
  - Revisit efficient \texttt{last}
  - Deriving memoization
  - Deriving the CPS translation
- Draw conclusions about general applicability
Example: Fetching the last element in a list in Haskell

\[
\begin{align*}
\text{last } [] & = \text{error } "\text{last: } []\text{"} \\
\text{last } [x] & = x \\
\text{last } (x:xs) & = \text{last } xs
\end{align*}
\]
Example: Fetching the last element in a list in Haskell

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

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

```haskell
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

\[
\text{last} :: [a] \rightarrow a \\
\text{last} = \lambda v \rightarrow \text{case } v \text{ of} \\
\hspace{1cm} [] \rightarrow \text{error "last: []"} \\
\hspace{1cm} (x:xs) \rightarrow \text{last_work } x \hspace{1mm} xs
\]

\[
\text{last_work} :: a \rightarrow [a] \rightarrow a \\
\text{last_work} = \lambda x \hspace{1mm} xs \rightarrow \text{case } xs \text{ of} \\
\hspace{1cm} [] \rightarrow x \\
\hspace{1cm} (x':xs') \rightarrow \text{last_work } x' \hspace{1mm} xs'
\]

- We have a \textit{wrapper} \text{last} which acts as an impedance match between \text{last} and \text{last_work}.
- We have a \textit{worker} \text{last_work} which does the recursive computation
- There now is only one \textit{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
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
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
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.
\( \beta \)-reduce worker

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

\[
\text{last}_\text{work} :: a \rightarrow [a] \rightarrow a \\
\text{last}_\text{work} = \lambda x \; xs \rightarrow \\
& \quad (\lambda v \rightarrow \text{case } v \text{ of} \\
& \quad & \quad [] \rightarrow \text{error "last: []"} \\
& \quad & \quad (x:xs) \rightarrow \text{case } xs \text{ of} \\
& \quad & \quad & \quad [] \rightarrow x \\
& \quad & \quad & \quad (_,_) \rightarrow \\
& \quad & \quad (\lambda v \rightarrow \text{case } v \text{ of} \\
& \quad & \quad & \quad [] \rightarrow \text{error "last: []"} \\
& \quad & \quad & \quad (x:xs) \rightarrow \text{last}_\text{work} x \; xs) \; xs) \; (x:xs)
\]

- We now perform \( \beta \)-reduction
\[\begin{align*} 
\text{last} &:: [a] \rightarrow a \\
\text{last} &= \lambda v \rightarrow \text{case } v \text{ of} \\
& \quad [] \rightarrow \text{error } "\text{last: []}" \\
& \quad (x:xs) \rightarrow \text{last\_work } x \; xs \\
\text{last\_work} &:: a \rightarrow [a] \rightarrow a \\
\text{last\_work} &= \lambda x \; xs \rightarrow \\
& \quad \text{case } (x:xs) \text{ of} \\
& \quad \quad [] \rightarrow \text{error } "\text{last: []}" \\
& \quad \quad (x:xs) \rightarrow \text{case } xs \text{ of} \\
& \quad \quad \quad [] \rightarrow x \\
& \quad \quad \quad (_:\_:_\_) \rightarrow \\
& \quad \quad \quad (\lambda v \rightarrow \text{case } v \text{ of} \\
& \quad \quad \quad \quad [] \rightarrow \text{error } "\text{last: []}" \\
& \quad \quad \quad \quad (x:xs) \rightarrow \text{last\_work } x \; xs) \; xs \\
\end{align*}\]

- We now perform \(\beta\)-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
\( \beta \)-reduce worker

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

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

We now perform \( \beta \)-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

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

\[
\text{last\_work :: } a \rightarrow [a] \rightarrow a \\
\text{last\_work } = \ \lambda \ x \ \text{xs} \rightarrow
\]

\[
\text{case } \text{xs of} \\
\quad [] \rightarrow x \\
\quad (x:xs) \rightarrow \ \text{last\_work } x \ \text{xs}
\]

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

last :: [a] → a

Recursion
wrap and unwrap

\[
\text{last :: } [a] \rightarrow a \quad \text{work :: } a \rightarrow [a] \rightarrow a
\]
wrap and unwrap

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

Recursion

wrap

Andy Gill, Graham Hutton
The Worker/Wrapper Transformation
February 6, 2008
wrap and unwrap in General

last :: $A$ Recursion
wrap and unwrap in General

\[ \text{last} :: A \quad \text{work} :: B \]

\[ \text{wrap} \quad \text{unwrap} \]
wrap and unwrap in General

\[
\text{last} :: A \quad \text{work} :: B
\]

Recursion

wrap
Key wrap and unwrap Diagram

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) {$A$};
  \node (B) at (1,0) {$B$};
  \draw[->] (A) to[bend left] node {unwrap} (B);
  \draw[->] (B) to[bend left] node {wrap} (A);
  \node at (-0.5,0) {last ::};
  \node at (1.5,0) {work ::};
\end{tikzpicture}
\end{center}
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 = idₐ  \quad (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 rolling rule

\[ \text{fix}(f . g) = f(\text{fix}(g . f)) \]

Intuitively correct because both sides expand into

\[ f(g(f(g(f(g(\ldots)))))), \]
Proof of Worker/Wrapper Theorem

\begin{align*}
\text{comp} \\
= \{ \text{applying} \text{comp} \} \\
\text{fix body} \\
= \{ \text{id is the identity for } \cdot \} \\
\text{fix (id } \cdot \text{ body)} \\
= \{ \text{assuming } \text{wrap} \cdot \text{unwrap} = \text{id} \} \\
\text{fix (wrap } \cdot \text{ unwrap } \cdot \text{ body)} \\
= \{ \text{rolling rule} \} \\
\text{wrap (fix (unwrap } \cdot \text{ body } \cdot \text{ wrap)}) \\
= \{ \text{define } \text{work} = \text{fix (unwrap } \cdot \text{ body } \cdot \text{ wrap}) \} \\
\text{wrap work}
\end{align*}
The worker/wrapper assumptions

Key step of proof

\[
\begin{align*}
\text{fix (id \cdot body)} & = \{ \text{assuming wrap \cdot unwrap = id} \} \\
\text{fix (wrap \cdot unwrap \cdot body)} & \\
\end{align*}
\]

We can actually use any of three different assumptions here

\[
\begin{align*}
\text{wrap \cdot unwrap} & = \text{id} & \text{(basic assumption)} \\
\downarrow \\
\text{wrap \cdot unwrap \cdot body} & = \text{body} & \text{(body assumption)} \\
\downarrow \\
\text{fix (wrap \cdot unwrap \cdot body)} & = \text{fix body} & \text{(fix-point assumption)}
\end{align*}
\]
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)

= \{ \beta-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 \cdot \}

(\ 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

\[
\text{wrap . unwrap . body}
\]

\[
= \{ \text{apply wrap, unwrap and } \cdot \} \\
= \{ \beta\text{-reductions} \}
\]

\[
(\ \text{fn} \rightarrow \) \\
(\ \text{xs} \rightarrow \text{case xs of} \\
\hspace{1em} [] \rightarrow \text{error "last: []"} \\
\hspace{1em} (x:\text{xs}) \rightarrow \text{case (x:\text{xs}) of} \\
\hspace{2em} [] \rightarrow \text{error "last: []"} \\
\hspace{2em} (x:\text{[]} \rightarrow x \\
\hspace{2em} (x:\text{xs}) \rightarrow \text{fn xs})
\]
Testing the body worker/wrapper assumption

wrap . unwrap . body

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

(\ fn \rightarrow
  (\ xs \rightarrow case xs of
    [] \rightarrow error "last: []"
    (x:xs) \rightarrow case xs of
      [] \rightarrow x
      xs \rightarrow fn xs))
Testing the body worker/wrapper assumption

wrap . unwrap . body

= \{ apply wrap, unwrap and \}
= \{ \beta\text{-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

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

\[
\text{work} :: a \rightarrow [a] \rightarrow a \\
\text{work} = \text{fix (unwrap . body . wrap)}
\]
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)
    )

Andy Gill, Graham Hutton

The Worker/Wrapper Transformation

February 6, 2008 24 / 31
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
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

\[ \text{fib} :: \text{Int} \to \text{Int} \]
\[ \text{fib} \ n = \]
\[ \ \ \ \text{if} \ n < 2 \]
\[ \ \ \ \ \ \ \text{then} \ 1 \]
\[ \ \ \ \\text{else} \ \text{fib} \ (n-1) + \text{fib} \ (n-2) \]

\[ \text{wrap} :: [\text{Nat}] \to (\text{Nat} \to \text{Nat}) \]
\[ \text{wrap} \ \text{xs} \ \text{ix} = \text{xs} \ !! \ \text{ix} \]

\[ \text{unwrap} :: (\text{Nat} \to \text{Nat}) \to [\text{Nat}] \]
\[ \text{unwrap} \ \text{f} = \text{map} \ \text{f} \ [0..] \]

After proving that \( \text{wrap} \circ \text{unwrap} = \text{id} \), and applying rewrites, we reach

\[ \text{fib} :: \text{Nat} \to \text{Nat} \]
\[ \text{fib} \ n = \text{work} \ !! \ n \]

\[ \text{work} = \text{map} \ \text{f} \ [0..] \]

\[ \text{where} \ \text{f} = \text{if} \ n < 2 \ \text{then} \ 1 \ \text{else} \ \text{work} \ !! \ (n-1) + \text{work} \ !! \ (n-2) \]
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
```
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

\[\text{work} :: \text{Expr} \to (\text{Int} \to \text{Maybe Int}) \to \text{Maybe Int} \to \text{Maybe Int} \]

\[\text{work (Val n)} \quad s \ f = s \ n\]
\[\text{work (Add x y)} \quad s \ f = \text{work x (} \lambda n \to \text{work y (} \lambda m \to s (n+m)) \text{) f) f}\]
\[\text{work (Throw)} \quad s \ f = f\]
\[\text{work (Catch x y)} \quad s \ f = \text{work x s (work y s f)}\]
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