The worker/wrapper transformation

Jaak Randmets

April 20, 2010

Introduction

- The worker/wrapper transformation is a technique for changing the type of computation.
- Usually the aim is to improve performance by improving the choice of data structures used.
- Well known to compiler writers, but not so in the functional programming community.
- In this talk we provide systematic recipe for its use and explore it using wide range of examples.

The basic idea

Given some (recursive) function:

$$f = \mathbf{body}$$

the first step is to apply appropriate functions *wrap* and *unwrap* that allow the function f to be redefined by equation $f = \mathit{wrap}\ (\mathit{unwrap}\ \mathbf{body})$. Next step is to split the function into two by naming the intermediate result:

$$\begin{aligned} f &= \mathit{wrap}\ \mathit{work} \\ \mathit{work} &= \mathit{unwrap}\ \mathbf{body} \end{aligned}$$

We then elimite the mutual recursion:

$$\begin{aligned} f &= \mathit{wrap}\ \mathit{work} \\ \mathit{work} &= \mathit{unwrap}\ (\mathbf{body}\ [\mathit{wrap}\ \mathit{work}\ /\ f\,]) \end{aligned}$$

The worker/wrapper transformation

- We begin by defining fixed point operator in Haskell:

$$fix :: (a \to a) \to a$$
$$fix\ f = f\ (fix\ f)$$

- In order to formalize the worker/wrapper transformation we will use property of fixed points known as the *rolling rule*:

$$fix\ (g \circ f) = g\ (fix\ (f \circ g))$$

Intuitively this is correct because both sides expand to the application $g\ (f\ (g\ (f \ldots)))$.

The worker/wrapper transformation

- Supposed we have computation defined as a fixed point:

$$\begin{aligned} &comp :: A \\ &comp = fix\ body \\ &body :: A \to A \end{aligned}$$

  We wish to change the underlying type A to some other type B.

- Worker/wrapper approach to this problem is to define conversion functions:

$$\begin{aligned} &unwrap :: A \to B \\ &wrap \quad :: B \to A \end{aligned}$$

  such that $wrap \circ unwrap = id$.

The worker/wrapper transformation

$$
\begin{aligned}
& comp \\
= & \quad \{ \text{ applying } comp \ \} \\
& fix\ body \\
= & \quad \{ \ id \text{ is identity for } \circ \ \} \\
& fix\ (id \circ body) \\
= & \quad \{ \text{ assuming } wrap \circ unwrap = id \ \} \\
& fix\ (wrap \circ unwrap \circ body) \\
= & \quad \{ \text{ use of rolling rule } \} \\
& wrap\ (fix\ (unwrap \circ body \circ wrap)) \\
= & \quad \{ \text{ define } work = fix\ (unwrap \circ body \circ wrap) \ \} \\
& wrap\ work
\end{aligned}
$$

The worker/wrapper transformation

Sometimes we might require a weaker property. Any of the following assumptions is valid:

Worker/wrapper assumptions

- $wrap \circ unwrap = id$
- $wrap \circ unwrap \circ body = body$
- $fix\ (wrap \circ unwrap \circ body) = fix\ body$

In general it's not the case that $unwrap \circ wrap = id$. However, following fusion property holds:

Worker/wrapper fusion

If $wrap \circ unwrap = id$, then $unwrap\ (wrap\ work) = work$.

Can easily be shown by taking:

$$work = unwrap\ (body\ (wrap\ work))$$

The worker/wrapper transformation

The worker/wrapper transformation

If $comp :: A$ is a recursive computation defined by $comp = fix\ body$ for some $body :: A \to A$, and $wrap :: B \to A$ and $unwrap :: A \to B$ are conversion functions satisfying any of the worker/wrapper assumptions, then:

$$comp = wrap\ work$$

where $work :: B$ is defined by:

$$work = fix\ (unwrap \circ body \circ wrap)$$

Difference lists

- Difference lists is an alternative way of representing lists:

$$\textbf{type}\ \mathit{DList}\ a = [a] \to [a]$$

- Naturally we need to convert to and from difference lists:

$$\begin{aligned}
&\mathit{fromList} && :: [a] \to \mathit{DList}\ a \\
&\mathit{fromList}\ xs && = (xs +\!\!+) \\
&\mathit{toList} && :: \mathit{DList}\ a \to [a] \\
&\mathit{toList}\ f && = f\ []
\end{aligned}$$

- We observe an identity $\mathit{toList} \circ \mathit{fromList} = \mathit{id}$:

$$(\mathit{toList} \circ \mathit{fromList})\ xs = xs +\!\!+ [\,] = xs$$

Difference lists

- Important property of difference lists is that *fromList* forms a morphism from lists to functions:

$$\begin{aligned} \mathit{fromList}\ (xs +\!\!+\ ys) &= \mathit{fromList}\ xs \circ \mathit{fromList}\ ys \\ \mathit{fromList}\ [\,] &= \mathit{id} \end{aligned}$$

- We can verify this by simple calculations:

$$\begin{aligned} \mathit{fromList}\ (xs +\!\!+\ ys)\ zs &= (xs +\!\!+\ ys) +\!\!+\ zs \\ &= xs +\!\!+\ (ys +\!\!+\ zs) \\ &= \mathit{fromList}\ xs\ (ys +\!\!+\ zs) \\ &= \mathit{fromList}\ xs\ (\mathit{fromList}\ ys\ zs) \\ &= (\mathit{fromList}\ xs \circ \mathit{fromList}\ ys)\ zs \end{aligned}$$

$$\begin{aligned} \mathit{fromList}\ [\,]\ zs &= [\,] +\!\!+\ zs \\ &= zs \end{aligned}$$

Reverse

- Let us consider the following definition of reverse:

$$\begin{aligned} &rev && :: [a] \rightarrow [a] \\ &rev\ [\,] && = [\,] \\ &rev\ (x : xs) && = rev\ xs +\!\!+ [x] \end{aligned}$$

- As a first step we redefine *rev* as a fixed point

$$\begin{aligned} &rev && :: [a] \rightarrow [a] \\ &rev && = fix\ body \\ &body && :: ([a] \rightarrow [a]) \rightarrow ([a] \rightarrow [a]) \\ &body\ f\ [\,] && = [\,] \\ &body\ f\ (x : xs) && = f\ xs +\!\!+ [x] \end{aligned}$$

Reverse

- Our aim is to change to computation type $[a] \to [a]$ to a type $[a] \to DList\ a$:

$$
\begin{aligned}
unwrap &:: ([a] \to [a]) \to ([a] \to DList\ a) \\
unwrap\ f &= fromList \circ f \\
wrap &:: ([a] \to DList\ a) \to ([a] \to [a]) \\
wrap\ g &= toList \circ g
\end{aligned}
$$

- We can verify the worker/wrapper assumption by:

$$
\begin{aligned}
(wrap \circ unwrap)\ f &= wrap\ (unwrap\ f) \\
&= toList \circ unwrap\ f \\
&= toList \circ fromList \circ f \\
&= id \circ f \\
&= f
\end{aligned}
$$

Reverse

Simplifying *rev*

$$
\begin{aligned}
rev &:: [a] \to [a] \\
rev &= wrap\ work \\
work &:: [a] \to DList\ a \\
work &= unwrap\ (body\ (wrap\ work))
\end{aligned}
$$

- Inline *wrap* in body of *rev*.
- η-expand *work*.

Reverse

Simplifying *rev*

$$
\begin{aligned}
rev &:: [a] \to [a] \\
rev\ xs &= work\ xs\ [\,] \\
work &:: [a] \to DList\ a \\
work\ xs &= unwrap\ (body\ (wrap\ work))\ xs
\end{aligned}
$$

- Expand *unwrap*.

Reverse

Simplifying *rev*

$$
\begin{aligned}
rev &:: [a] \rightarrow [a] \\
rev\ xs &= work\ xs\ [\,] \\
work &:: [a] \rightarrow DList\ a \\
work\ xs &= fromList\ (body\ (wrap\ work)\ xs)
\end{aligned}
$$

- Inline *body*.

Reverse

Simplifying *rev*

$$
\begin{array}{lll}
rev & :: & [a] \to [a] \\
rev\ xs & = & work\ xs\ [\,] \\
work & :: & [a] \to DList\ a \\
work\ xs & = & fromList\ (\textbf{case}\ xs\ \textbf{of} \\
 & & \quad [\,] \qquad\quad \to [\,] \\
 & & \quad (x : xs) \to wrap\ work\ xs +\!\!+ [x])
\end{array}
$$

- case transformation.

Reverse

Simplifying *rev*

$$\begin{aligned}
&rev &&:: [a] \rightarrow [a] \\
&rev\ xs &&= work\ xs\ [\,] \\
&work &&:: [a] \rightarrow DList\ a \\
&work\ [\,] &&= fromList\ [\,] \\
&work\ (x : xs) &&= fromList\ (wrap\ work\ xs +\!\!+ [x])
\end{aligned}$$

- *fromList* is morphism.

Reverse

Simplifying *rev*

$$
\begin{aligned}
&rev &&:: [a] \to [a] \\
&rev\ xs &&= work\ xs\ [\,] \\
&work &&:: [a] \to DList\ a \\
&work\ [\,] &&= id \\
&work\ (x : xs) &&= fromList\ (wrap\ work\ xs) \circ fromList\ [x]
\end{aligned}
$$

- $$
\begin{aligned}
&\ fromList\ (wrap\ work\ xs) \\
&= unwrap\ (wrap\ work)\ xs \\
&= work\ xs
\end{aligned}
$$

Reverse

Simplifying *rev*

$$
\begin{aligned}
&\mathit{rev} && :: [a] \rightarrow [a] \\
&\mathit{rev}\ \mathit{xs} && = \mathit{work}\ \mathit{xs}\ [\,] \\
&\mathit{work} && :: [a] \rightarrow \mathit{DList}\ a \\
&\mathit{work}\ [\,] && = \mathit{id} \\
&\mathit{work}\ (x : \mathit{xs}) && = \mathit{work}\ \mathit{xs} \circ \mathit{fromList}\ [x]
\end{aligned}
$$

- η-expand *work*.
- Expand *fromList* and $\circ$.

Reverse

Simplifying *rev*

$$
\begin{aligned}
&rev && :: [a] \to [a] \\
&rev\ xs && = work\ xs\ [\,] \\
&work && :: [a] \to [a] \to [a] \\
&work\ [\,]\ ys && = ys \\
&work\ (x : xs)\ ys && = work\ xs\ (x : ys)
\end{aligned}
$$

- We have reached linear time accumulating version of list reversing function.

Memoisation

Our approach to memoising is to observe that any function f from natural numbers can be represented as infinite stream $[f\ 0, f\ 1, \ldots]$.

unwrap/wrap

$$
\begin{aligned}
&\mathit{unwrap} && :: (\mathit{Nat} \to a) \to \mathit{Stream}\ a \\
&\mathit{unwrap}\ f && = \mathit{map}\ f\ [0\,..] \\
& && = f\ 0 : \mathit{unwrap}\ (f \circ (+1)) \\
&\mathit{wrap} && :: \mathit{Stream}\ a \to (\mathit{Nat} \to a) \\
&\mathit{wrap}\ \mathit{xs} && = (\mathit{xs}!!)
\end{aligned}
$$

$$
\begin{aligned}
&(!!) && :: \mathit{Stream}\ a \to \mathit{Nat} \to a \\
&\mathit{xs}\ !!\ 0 && = \mathit{head}\ \mathit{xs} \\
&\mathit{xs}\ !!\ (n+1) && = (\mathit{tail}\ \mathit{xs})\ !!\ n
\end{aligned}
$$

Memoisation

We will show that $wrap \circ unwrap = id$ by expanding $wrap$ and making arguments explicit: $(unwrap\ f)\ !!\ n = f\ n$.

- Base case:

$$
\begin{aligned}
& (unwrap\ f)\ !!\ 0 \\
= & (f\ 0 : unwrap\ (f \circ (+1)))\ !!\ 0 \\
= & f\ 0
\end{aligned}
$$

- Inductive case:

$$
\begin{aligned}
& (unwrap\ f)\ !!\ (n+1) \\
= & (f\ 0 : unwrap\ (f \circ (+1)))\ !!\ (n+1) \\
= & unwrap\ (f \circ (+1))\ !!\ n \\
= & (f \circ (+1))\ n \\
= & f\ (n+1)
\end{aligned}
$$

Fibonacci

Fibonacci function

$$
\begin{aligned}
&\mathit{fib} &&:: \mathit{Nat} \to \mathit{Nat}\\
&\mathit{fib} &&= \mathit{fix}\ \mathit{body}\\
&\mathit{body} &&:: (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{body}\ f\ 0 &&= 0\\
&\mathit{body}\ f\ 1 &&= 1\\
&\mathit{body}\ f\ (n+2) &&= f\ n + f\ (n+1)
\end{aligned}
$$

Fibonacci

Simplifying *fib*

$$\begin{aligned} fib &:: Nat \to Nat \\ fib &= wrap\ work \\ work &:: Stream\ Nat \\ work &= unwrap\ (body\ (wrap\ work)) \end{aligned}$$

- Apply *wrap* in *fib*.
- Apply *unwrap* in *work*.

Fibonacci

Simplifying *fib*

$$\begin{aligned} fib &:: Nat \rightarrow Nat \\ fib\ n &= work\ !!\ n \end{aligned}$$

$$\begin{aligned} work &:: Stream\ Nat \\ work &= map\ (body\ (wrap\ work))\ [0\,..] \end{aligned}$$

- Inline *body*.

Fibonacci

Simplifying *fib*

$$
\begin{aligned}
&\mathit{fib} && :: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{fib}\ n && = \mathit{work}\ !!\ n \\
&\mathit{work} && :: \mathit{Stream}\ \mathit{Nat} \\
&\mathit{work} && = \mathit{map}\ (\lambda n \to \textbf{case}\ n\ \textbf{of} \\
&\quad 0 && \to 0 \\
&\quad 1 && \to 1 \\
&\quad (n+2) && \to \mathit{wrap}\ \mathit{work}\ n + \mathit{wrap}\ \mathit{work}\ (n+1))\ [0\,..]
\end{aligned}
$$

- Apply *wrap*.

Fibonacci

Simplifying *fib*

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{fib}\ n = \mathit{work}\ !!\ n \\
&\mathit{work} :: \mathit{Stream}\ \mathit{Nat} \\
&\mathit{work} = \mathit{map}\ (\lambda n \to \mathbf{case}\ n\ \mathbf{of} \\
&\quad 0 \to 0 \\
&\quad 1 \to 1 \\
&\quad (n+2) \to \mathit{work}\ !!\ n + \mathit{work}\ !!\ (n+1))\ [0\,..]
\end{aligned}
$$

- Finally introduce f.

Fibonacci

Simplifying *fib*

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Nat} \rightarrow \mathit{Nat} \\
&\mathit{fib}\ n = \mathit{work}\ !!\ n \\
&\mathit{work} :: \mathit{Stream}\ \mathit{Nat} \\
&\mathit{work} = \mathit{map}\ f\ [0\,..] \\
&\quad \textbf{where} \\
&\qquad f\ 0 = 0 \\
&\qquad f\ 1 = 1 \\
&\qquad f\ (n+2) = \mathit{work}\ !!\ n + \mathit{work}\ !!\ (n+1)
\end{aligned}
$$

- We have reached a quadratic Fibonacci function from exponential one.

Continuations

- Any type can be alternatively represented as a continuation. Idea is to represent value x as a function $\lambda c \rightarrow c\ x$ that takes a (continuation) c and applies it to x.

$$\textbf{type}\ Cont\ a = (a \rightarrow a) \rightarrow a$$

- We can convert from and to continuations as follows:

$$\begin{aligned} &toCont &&:: a \rightarrow Cont\ a \\ &toCont\ x &&= \lambda c \rightarrow c\ x \\ &fromCont &&:: Cont\ a \rightarrow a \\ &fromCont\ f &&= f\ id \end{aligned}$$

- It's easy to show that:

$$\begin{aligned} (fromCont \circ toCont)\ x &= (toCont\ x)\ id \\ &= (\lambda c \rightarrow c\ x)\ id \\ &= x \end{aligned}$$

Evaluation

- We will now consider simple expression language:

$$
\begin{aligned}
\textbf{data } Expr &= Val\ Int \\
&\mid Expr \oplus Expr \\
&\mid Throw \\
&\mid Catch\ Expr\ Expr
\end{aligned}
$$

- With standard evaluation function:

$$
\begin{array}{ll}
eval & :: Expr \rightarrow MInt \\
eval\ (Val\ n) & = Just\ n \\
eval\ (e_0 \oplus e_1) & = \textbf{case } eval\ e_0 \textbf{ of} \\
& \quad Nothing \rightarrow Nothing \\
& \quad Just\ n \rightarrow \textbf{case } eval\ e_1 \textbf{ of} \\
& \quad\quad Nothing \rightarrow Nothing \\
& \quad\quad Just\ m \rightarrow Just\ (n + m) \\
eval\ Throw & = Nothing \\
eval\ (Catch\ e_0\ e_1) & = \textbf{case } eval\ e_0 \textbf{ of} \\
& \quad Nothing \rightarrow eval\ e_1 \\
& \quad Just\ n \rightarrow Just\ n
\end{array}
$$

Evaluation

- One might expect to move to representation of type $Expr \to Cont\ MInt$. Instead we will have different continuation for exceptional control flow and regular control flow.
- We split $MInt \to MInt$ into two and reach type:

$$Expr \to (Int \to MInt) \to MInt \to MInt$$

- Wrap/unwrap are given by:

$$\begin{aligned}
unwrap\ g\ e\ s\ f &= \textbf{case}\ g\ e\ \textbf{of} \\
&\qquad Nothing \to f \\
&\qquad Just\ n \quad \to s\ n \\
wrap\ h\ e &= h\ e\ Just\ Nothing
\end{aligned}$$

Evaluation

- Worker/wrapper assumption can easily verified as follows:

$$
\begin{aligned}
& (\mathit{wrap} \circ \mathit{unwrap})\ g\ e \\
=\ & \mathit{wrap}\ (\mathit{unwrap}\ g)\ e \\
=\ & \mathit{unwrap}\ g\ e\ \mathit{Just}\ \mathit{Nothing} \\
=\ & \textbf{case}\ g\ e\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathit{Just}\ n \\
=\ & g\ e
\end{aligned}
$$

- We can now apply the worker/wrapper transformation.

Evaluation

Simplifying *eval*

$$\begin{aligned} eval &:: Expr \rightarrow MInt \\ eval &= wrap\ work \end{aligned}$$

$$\begin{aligned} work &:: Expr \rightarrow (Int \rightarrow MInt) \rightarrow MInt \rightarrow MInt \\ work &= unwrap\ (body\ (wrap\ work)) \end{aligned}$$

- Inline *wrap*.
- η-expand work.

Evaluation

Simplifying *eval*

$$\begin{aligned}
eval &:: Expr \rightarrow MInt \\
eval &= work\ e\ Just\ Nothing \\
work &:: Expr \rightarrow (Int \rightarrow MInt) \rightarrow MInt \rightarrow MInt \\
work\ e\ s\ f &= unwrap\ (body\ (wrap\ work))\ e\ s\ f
\end{aligned}$$

- Apply *unwrap*.

Evaluation

Simplifying *eval*

$$
\begin{aligned}
&eval &&:: Expr \rightarrow MInt \\
&eval &&= work\ e\ Just\ Nothing \\
&work &&:: Expr \rightarrow (Int \rightarrow MInt) \rightarrow MInt \rightarrow MInt \\
&work\ e\ s\ f &&= \textbf{case}\ body\ (wrap\ work)\ e\ \textbf{of} \\
& && \qquad Nothing \rightarrow f \\
& && \qquad Just\ n \quad \rightarrow s\ n
\end{aligned}
$$

- Apply *body*.

Evaluation

Simplifying *eval*

$$
\begin{array}{l}
work\ e\ s\ f = \textbf{case}\ (\textbf{case}\ e\ \textbf{of} \\
\quad Val\ n \to Just\ n \\
\quad e_0 \oplus e_1 \to \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\qquad Nothing \to Nothing \\
\qquad Just\ n \to \textbf{case}\ wrap\ work\ e_1\ \textbf{of} \\
\qquad\quad Nothing \to Nothing \\
\qquad\quad Just\ m \to Just\ (n + m) \\
\quad Throw \to Nothing \\
\quad Catch\ e_0\ e_1 \to \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\qquad Nothing \to wrap\ work\ e_1 \\
\qquad Just\ n \to Just\ n)\ \textbf{of} \\
\quad Nothing \to f \\
\quad Just\ n \to s\ n
\end{array}
$$

Evaluation

Simplifying *eval*

$$
\begin{array}{l}
work\ e\ s\ f = \textbf{case}\ e\ \textbf{of} \\
\quad Val\ n \qquad\quad \rightarrow s\ n \\
\quad e_0 \oplus e_1 \qquad \rightarrow \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\qquad Nothing \rightarrow f \\
\qquad Just\ n \quad \rightarrow \textbf{case}\ wrap\ work\ e_1\ \textbf{of} \\
\qquad\quad Nothing \rightarrow f \\
\qquad\quad Just\ m \rightarrow s\ (n + m) \\
\quad Throw \qquad\ \rightarrow f \\
\quad Catch\ e_0\ e_1 \rightarrow \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\qquad Nothing \rightarrow \textbf{case}\ wrap\ work\ e_1\ \textbf{of} \\
\qquad\quad Nothing \rightarrow f \\
\qquad\quad Just\ n \rightarrow s\ n \\
\qquad Just\ n \rightarrow s\ n
\end{array}
$$

Evaluation

Simplifying *eval*

$$
\begin{array}{ll}
work\ (Val\ n)\ s\ f & = s\ n \\
work\ (e_0 \oplus e_1)\ s\ f & = \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\quad Nothing \rightarrow f & \\
\quad Just\ n \quad \rightarrow \textbf{case}\ wrap\ work\ e_1\ \textbf{of} & \\
\qquad Nothing \rightarrow f & \\
\qquad Just\ m \rightarrow s\ (n + m) & \\
work\ Throw\ s\ f & = f \\
work\ (Catch\ e_0\ e_1)\ s\ f & = \textbf{case}\ wrap\ work\ e_0\ \textbf{of} \\
\quad Nothing \rightarrow \textbf{case}\ wrap\ work\ e_1\ \textbf{of} & \\
\qquad Nothing \rightarrow f & \\
\qquad Just\ n \rightarrow s\ n & \\
\quad Just\ n \rightarrow s\ n &
\end{array}
$$

Evaluation

- By re-exposing *unwrap* we can perform further simplification:

$$
\begin{aligned}
& \textbf{case}\ \mathit{wrap}\ \mathit{work}\ x\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow g \\
& \quad \mathit{Just}\ n \rightarrow s\ n \\
= & \qquad \{\ \text{unapply}\ \mathit{unwrap}\ \} \\
& \mathit{unwrap}\ (\mathit{wrap}\ \mathit{work})\ x\ s\ g \\
= & \qquad \{\ \text{worker/wrapper fusion}\ \} \\
& \mathit{work}\ x\ s\ g
\end{aligned}
$$

- We can apply this multiple times.

Evaluation

Simplifying *eval*

$$
\begin{aligned}
&work\ (Val\ n)\ s\ f &&= s\ n \\
&work\ (e_0 \oplus e_1)\ s\ f &&= work\ e_0\ (\lambda n \rightarrow \\
& && \quad work\ e_1\ (\lambda m \rightarrow s\ (n + m))\ f)\ f \\
&work\ Throw\ s\ f &&= f \\
&work\ (Catch\ e_0\ e_1)\ s\ f &&= work\ e_0\ s\ (work\ e_1\ s\ f)
\end{aligned}
$$

- Corresponds to abstract machine that works on two stacks, one for normal evaluation and other for handling exceptions.

Fin.