JFP **19** (2): 227–251, 2009. © 2009 Cambridge University Press
doi:10.1017/S0956796809007175 Printed in the United Kingdom

The worker/wrapper transformation

ANDY GILL
University of Kansas, USA
(e-mail: andygill@ku.edu)

GRAHAM HUTTON
University of Nottingham, UK
(e-mail: gmh@cs.nott.ac.uk)

Abstract

The worker/wrapper transformation is a technique for changing the type of a computation, usually with the aim of improving its performance. It has been used by compiler writers for many years, but the technique is little known in the wider functional programming community, and has never been described precisely. In this article we explain, formalise and explore the generality of the worker/wrapper transformation. We also provide a systematic recipe for its use as an equational reasoning technique for improving the performance of programs, and illustrate the power of this recipe using a range of examples.

1 Introduction

The worker/wrapper transformation is a technique for transforming a computation of one type into a *worker* of a different type, together with a *wrapper* that acts as an impedance matcher between the original and new computations.

The technique can be used to improve the performance of functional programs by improving the choice of data structures used. For example, in the Glasgow Haskell Compiler, a specific instance of the worker/wrapper transformation is used to replace boxed data structures by unboxed structures, based upon strictness information (Peyton Jones & Launchbury, 1991). Despite its practical importance, however, the technique is little known in the wider functional programming community, and to the best of our knowledge has never been described precisely.

In this article we explain, formalise, and explore the generality of the worker/wrapper transformation. We also provide a systematic recipe for its use, and illustrate the power of this recipe using a range of programming examples. More precisely, the article makes the following contributions:

- We present the worker/wrapper transformation in a general semantic framework (Section 3), rather than via specific syntactic instances, in order to make the technique as widely applicable as possible.
- We give a correctness proof for the technique (Section 3), based upon the use of the rolling rule from fixed point calculus, which provides a range of explicit conditions under which the technique can be applied.

- We provide a systematic recipe for its use as an equational reasoning technique for improving the performance of programs (Section 3), by means of a step-by-step process for applying the transformation.
- We apply this recipe to four different approaches to program optimisation, based upon the use of accumulation (Section 4), unboxing (Section 5), memoisation (Section 6) and continuations (Section 7).

A summary of related work, and directions for further work, are provided in the concluding sections. The article is aimed at readers who are familiar with the basics of reasoning about functional programs, say to the level of Bird (1998), but no previous experience with the worker/wrapper transformation is assumed. The techniques are presented using Haskell (Peyton Jones, 2003), but can easily be adapted to other functional languages. An extended version of the article that includes all the proofs is available from the authors' web pages.

2 The basic idea

In this section we review the basic idea of the worker/wrapper transformation, in a similar manner to which it was originally described by Peyton Jones and Launchbury (1991), and implemented in the Glasgow Haskell Compiler.

Suppose that we are given a function f, defined in the form

$$f = \texttt{body}$$

where body is the right-hand side of the definition, and may include recursive calls to f. At this syntactic level, the first step in applying the worker/wrapper transformation is to define appropriate functions *wrap* and *unwrap* that allow the function f to be redefined by the equation $f = \mathit{wrap}\ (\mathit{unwrap}\ \texttt{body})$, which is then split into two equations by naming the intermediate result, as follows:

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

In this manner, f has been factorised into the application of a 'wrapper' function *wrap* to a 'worker' function *work*, itself defined by applying *unwrap* to the body of the original definition for f. Note that if f was originally recursive, then f and *work* are now mutually recursive. The next step in the process is to eliminate such mutual recursion by inlining the new definition for f in the definition for *work*, thereby making *work* into a recursive definition that is independent of f:

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

As usual, $e\,[e'/x]$ denotes the result of substituting e' for all free occurrences of the variable x in the expression e. The final step is then to simplify the resulting definitions for f and *work* using standard program transformation techniques.

For example, Peyton Jones and Launchbury (1991) show how this process can be used to transform a recursive definition of the factorial function into a more efficient version that is defined in terms of a worker that only uses unboxed integers, together with a wrapper that takes care of the initial unboxing and final boxing.

The above, syntactic, description of the worker/wrapper transformation is appealingly simple, but raises a number of important questions. Is the technique actually correct? How can this be proved? Under what conditions does the proof hold? How should it be used in practice? What kind of applications is it suitable for? Addressing these questions is the aim of the remainder of this article.

3 The worker/wrapper transformation

In order to give a precise treatment of the worker/wrapper transformation, and hence prove its correctness, we move from the informal syntactic approach based upon inlining to a formal semantic approach based upon the use of fixed points. We begin by defining a fixed point operator in Haskell:

$$\begin{aligned} &\mathit{fix} &&:: (a \to a) \to a \\ &\mathit{fix}\ f &&= f\ (\mathit{fix}\ f) \end{aligned}$$

The property of this operator that we will use to formalise the worker/wrapper transformation is the *rolling rule* (Backhouse, 2002), which allows us to pull the first argument of a composition outside a fixed point, resulting in the composition swapping the order of its arguments, or 'rolling over':

$$\mathit{fix}\ (g \circ f) = g\ (\mathit{fix}\ (f \circ g))$$

Informally, this rule is valid because both sides expand to the infinite application $g\ (f\ (g\ (f\ (g\ (f\ \ldots)))))$. A formal proof, together with a brief review of the fixed point approach to recursion, is provided in the Appendix.

Now consider the problem of changing the type of a recursive computation. More precisely, suppose that we are given a computation $\mathit{comp} :: A$, defined as the least fixed point $\mathit{comp} = \mathit{fix}\ \mathit{body}$ of some function $\mathit{body} :: A \to A$, and that we wish to change the underlying type from A to some other type B.

The worker/wrapper approach to this problem of changing types is based upon defining conversion functions $\mathit{unwrap} :: A \to B$ and $\mathit{wrap} :: B \to A$ such that the equation $\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_A$ holds, where id_A is the identity function for the type A. (To aid the reader, identity functions are subscripted with their types throughout the article.) This equation states that converting a value of the original type into the new type and then back again does not change the value, and formalises the idea that the type A can be faithfully represented by the type B.

Given such a setting, we can now calculate as follows:

$$\begin{aligned} & \mathit{comp} \\ = \quad & \quad \{\ \text{applying } \mathit{comp}\ \} \\ & \mathit{fix}\ \mathit{body} \\ = \quad & \quad \{\ \mathit{id} \text{ is the identity for } \circ\ \} \\ & \mathit{fix}\ (\mathit{id}_A \circ \mathit{body}) \end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{\ \dagger \text{ assuming } \mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_A\ \} \\
 & \mathit{fix}\ (\mathit{wrap} \circ \mathit{unwrap} \circ \mathit{body}) \\
= & \quad \{\ \text{rolling rule}\ \} \\
 & \mathit{wrap}\ (\mathit{fix}\ (\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap})) \\
= & \quad \{\ \text{define } \mathit{work} = \mathit{fix}\ (\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap})\ \} \\
 & \mathit{wrap}\ \mathit{work}
\end{array}
$$

That is, we have shown how an arbitrary recursive computation of type A can be factorised as a wrapper of type $B \to A$ applied to a new recursive worker computation of type B, based upon the assumption that the identity function for A can be split into the composition of two conversion functions. The following diagram summarises the primary typing relationships:

$$\begin{tikzcd}
\mathit{comp} :: \boxed{A} \arrow[rr, bend left, "\mathit{unwrap}"] & & \mathit{work} :: \boxed{B} \arrow[ll, bend left, "\mathit{wrap}"]
\end{tikzcd}$$

As we shall see, however, in general there is no requirement that this diagram commutes. For example, for some applications the assumption $\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_A$ may not be true in general, but only in the context (labelled by †) in which it is used in the above calculation. In particular, we may sometimes require the weaker property $\mathit{wrap} \circ \mathit{unwrap} \circ \mathit{body} = \mathit{id}_A \circ \mathit{body}$, which states that the assumption is true for values produced by *body*, or even $\mathit{fix}\ (\mathit{wrap} \circ \mathit{unwrap} \circ \mathit{body}) = \mathit{fix}\ (\mathit{id}_A \circ \mathit{body})$, which also takes the recursive context into account. That is, we have the following hierarchy of *worker/wrapper assumptions* that support the above calculation:

$$\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_A$$

$$\Downarrow$$

$$\mathit{wrap} \circ \mathit{unwrap} \circ \mathit{body} = \mathit{body}$$

$$\Downarrow$$

$$\mathit{fix}\ (\mathit{wrap} \circ \mathit{unwrap} \circ \mathit{body}) = \mathit{fix}\ \mathit{body}$$

For any given application, we will use the strongest such assumption that is valid, i.e. the simplest to verify; often the basic equation $\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_A$ will suffice. In conclusion, the original syntactic description of the worker/wrapper transformation can now be formalised on semantic grounds as shown in Figure 1.

Once this transformation has been applied, one then attempts to simplify the worker using normal fold/unfold transformation (Burstall & Darlington, 1977). It is usually convenient to begin by rewriting the worker using explicit recursion

$$\mathit{work} = \mathit{unwrap}\ (\mathit{body}\ (\mathit{wrap}\ \mathit{work}))$$

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

$$comp \quad = \quad wrap\ work$$

where $work :: B$ is defined by

$$work \quad = \quad \mathit{fix}\ (unwrap \circ body \circ wrap)$$

Fig. 1. The worker/wrapper transformation.

which transformation can be verified as follows:

$$\begin{array}{cl} & work \\ = & \quad \{\ \text{applying } work\ \} \\ & \mathit{fix}\ (unwrap \circ body \circ wrap) \\ = & \quad \{\ \text{applying } \mathit{fix}\ \} \\ & (unwrap \circ body \circ wrap)\ (\mathit{fix}\ (unwrap \circ body \circ wrap)) \\ = & \quad \{\ \text{unapplying } work\ \} \\ & (unwrap \circ body \circ wrap)\ work \\ = & \quad \{\ \text{applying } \circ\ \} \\ & unwrap\ (body\ (wrap\ work)) \end{array}$$

Further simplification of the worker is then typically driven by the desire to fuse together instances of *unwrap* and *wrap*, to eliminate the overhead of repeatedly converting between the two types. In general, it is not the case that $unwrap \circ wrap$ can be fused to give id_B, but the following *worker/wrapper fusion* property, in which the argument value of type B is the worker itself, is often applicable:

$$\text{If } wrap \circ unwrap = id_A\text{, then } unwrap\ (wrap\ work) = work$$

For example, this property will be utilised in two of our programming applications (Sections 4 and 7). The fusion property itself can be verified as follows:

$$\begin{array}{cl} & unwrap\ (wrap\ work) \\ = & \quad \{\ \text{applying } work\ \} \\ & unwrap\ (wrap\ (unwrap\ (body\ (wrap\ work)))) \\ = & \quad \{\ \text{assuming } wrap \circ unwrap = id_A\ \} \\ & unwrap\ (body\ (wrap\ work)) \\ = & \quad \{\ \text{unapplying } work\ \} \\ & work \end{array}$$

In summary, we have the following general recipe for applying the worker/wrapper transformation to change the type of a recursive computation:

- Express the computation as the least fixed point;
- Choose the desired new type for the computation;

- Define conversions between the original and new types;
- Check that they satisfy one of the worker/wrapper assumptions;
- Apply the worker/wrapper transformation;
- Simplify the resulting definitions.

We conclude this section by noting that the worker/wrapper transformation can also be formalised using *fixed point fusion* (Meijer *et al.*, 1991) which requires the additional property that *wrap* is strict ($wrap\ \bot = \bot$). Fortunately this property follows automatically from the basic worker/wrapper assumption $wrap \circ unwrap = id$, by virtue of the fact that any (monotonic) function that is right invertible must be strict. We prefer the formulation using the rolling rule because it corresponds directly to our understanding of why the transformation is valid.

4 Example: the reverse function

As a first example of our worker/wrapper recipe, we will use it to transform a simple definition for the function that reverses a list into a more efficient version that uses *accumulation*. This transformation is normally achieved using more elementary techniques (Hutton, 2007, Section 13.6), but we now show that it also fits naturally into our general worker/wrapper paradigm.

4.1 Hughes lists

We begin by reviewing the idea of representing lists using functions, which is the essential algorithmic idea underlying this example.

As observed by Hughes (1986), it is possible to represent the normal type of lists in an alternative manner using functions, which represent a list *xs* by the function (xs ++) that prepends *xs* to its argument. We can implement this idea by defining a type $H\ a$ of *Hughes lists*, together with a representation function *rep*

$$\begin{aligned} &\textbf{type}\ H\ a &&= [a] \to [a] \\ &rep &&:: [a] \to H\ a \\ &rep\ xs &&= (xs\ +\!\!+) \end{aligned}$$

An important property of this representation is that the function *rep* forms a homomorphism from lists to functions, in the sense that

$$\begin{aligned} rep\ (xs +\!\!+ ys) &= rep\ xs \circ rep\ ys \\ rep\ [\,] &= id_{[a]} \end{aligned}$$

That is, *rep* is a homomorphism from the monoid of (finite) lists, for which the associative operator is ++ and the unit is [], to the monoid of functions on lists, for which the operator is composition $\circ$ and the unit is the identity function $id_{[a]}$.

In addition to the function *rep* that converts normal lists into Hughes lists, it is also natural to consider conversion in the opposite direction, which is achieved by simply applying the given function to the empty list

$$\begin{aligned}
&abs && :: H\ a \to [a]\\
&abs\ f && = f\ []
\end{aligned}$$

This definition ensures that converting a list into Hughes form and back again returns the original list, or more formally, it ensures that $abs \circ rep = id_{[a]}$.

In summary, using the terminology of data representation (Hoare, 1972), we have shown how the 'abstract' type $[a]$ can be represented using the 'concrete' type $H\ a$. As is usual in this setting, note that the representation function *rep* is injective but not surjective, while the abstraction function *abs* is surjective but not injective.

4.2 Reverse

Now consider the following definition for the function that reverses a list:

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

Because of the use of $\mathbin{+\!\!+}$, this definition takes quadratic time in the length of its argument. We now show how this definition can be transformed into a more efficient *worker* that uses an extra argument to accumulate the result, together with a *wrapper* that takes care of the initial setup. The first step is to redefine *rev* as the least fixed point, by abstracting over the recursive call in the body

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

Following our worker/wrapper recipe, and utilising the idea of Hughes lists, our aim now is to transform the computation *rev* of type $[a] \to [a]$ into a worker of type $[a] \to H\ a$ (which abbreviates $[a] \to [a] \to [a]$ and hence introduces the necessary extra argument list), as illustrated by the following diagram:

$$\begin{tikzcd}
rev :: \boxed{[a] \to [a]} \arrow[r, bend left, "unwrap"] & work :: \boxed{[a] \to H\ a} \arrow[l, bend left, "wrap"]
\end{tikzcd}$$

Implementing the conversion functions *unwrap* and *wrap* is simply a matter of composing the conversion functions *rep* and *abs* from the previous section

$$\begin{aligned}
&\mathit{unwrap} &&:: ([a] \to [a]) \to ([a] \to H\ a)\\
&\mathit{unwrap}\ f &&= \mathit{rep} \circ f\\
&\mathit{wrap} &&:: ([a] \to H\ a) \to ([a] \to [a])\\
&\mathit{wrap}\ g &&= \mathit{abs} \circ g
\end{aligned}$$

Verifying the worker/wrapper assumption, in this case $\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_{[a]\to[a]}$, follows from the fact that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{[a]}$ by straightforward calculation.

Now that we have satisfied the preconditions for the worker/wrapper transformation, applying this transformation and rewriting the worker using explicit recursion rather than the *fix* operator gives the following result:

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

Making the list argument explicit in the definition for *rev*, and expanding out *wrap*, we obtain the wrapper for the efficient version of *rev*, which simply supplies the empty list as the initial accumulator

$$\begin{aligned}
&\mathit{rev} &&:: [a] \to [a]\\
&\mathit{rev}\ \mathit{xs} &&= \mathit{work}\ \mathit{xs}\ [\,]
\end{aligned}$$

Now let us focus our attention on simplifying the definition for the worker function. First of all, we redefine *work* using pattern matching

$$\begin{aligned}
&\mathit{work}\ [\,] &&= \mathit{rep}\ [\,]\\
&\mathit{work}\ (x : \mathit{xs}) &&= \mathit{rep}\ (\mathit{wrap}\ \mathit{work}\ \mathit{xs} +\!\!+ [x])
\end{aligned}$$

and the transformation can be verified as follows:

$$\begin{aligned}
&\quad \mathit{work}\ \mathit{xs}\\
=&\quad\quad \{\ \text{applying } \mathit{work}\ \}\\
&\quad \mathit{unwrap}\ (\mathit{body}\ (\mathit{wrap}\ \mathit{work}))\ \mathit{xs}\\
=&\quad\quad \{\ \text{applying } \mathit{unwrap}\ \}\\
&\quad \mathit{rep}\ (\mathit{body}\ (\mathit{wrap}\ \mathit{work})\ \mathit{xs})\\
=&\quad\quad \{\ \text{applying } \mathit{body}\ \}\\
&\quad \mathit{rep}\ (\mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\qquad\qquad [\,] \to [\,]\\
&\qquad\qquad (x : \mathit{xs}) \to \mathit{wrap}\ \mathit{work}\ \mathit{xs} +\!\!+ [x])\\
=&\quad\quad \{\ \text{distribution over } \mathbf{case}\ \}\\
&\quad \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\qquad [\,] \to \mathit{rep}\ [\,]\\
&\qquad (x : \mathit{xs}) \to \mathit{rep}\ (\mathit{wrap}\ \mathit{work}\ \mathit{xs} +\!\!+ [x])
\end{aligned}$$

The distribution step assumes that *rev* is only applied to finite lists, to ensure that the argument list *xs* in this step is not undefined ($\perp$).

Now we exploit the fact that $rep :: [a] \to H\ a$ is a homomorphism from lists to functions, to rewrite the new definition as follows:

$$\begin{aligned} &work\ [\,] &&= id_{[a]} \\ &work\ (x : xs) &&= rep\ (wrap\ work\ xs) \circ rep\ [x] \end{aligned}$$

Then we simplify the first argument of the composition in this definition by exposing the implicit use of *unwrap* and then fusing this with *wrap*

$$\begin{aligned} & rep\ (wrap\ work\ xs) \\ = \quad & \quad \{\ \text{unapplying } unwrap\ \} \\ & unwrap\ (wrap\ work)\ xs \\ = \quad & \quad \{\ \text{worker/wrapper fusion}\ \} \\ & work\ xs \end{aligned}$$

That is, we now have the following definition:

$$\begin{aligned} &work\ [\,] &&= id_{[a]} \\ &work\ (x : xs) &&= work\ xs \circ rep\ [x] \end{aligned}$$

Finally, if we make the extra list argument in this definition explicit, and then expand out $\circ$ and *rep*, we obtain the worker for the efficient version of reverse, which uses an extra list argument to accumulate the final result

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

We conclude this section with three observations about the above derivation. First of all, note that everything follows in a straightforward manner from the idea of using Hughes' representation of lists, in the sense that once we have made this decision, the rest of the derivation proceeds using standard fold/unfold transformation. Secondly, the derivation does not require the use of induction, other than the implicit use to verify the auxiliary result that finite lists form a monoid, which is used in proof that *rep* is a homomorphism. And finally, it is possible to generalise from this example and formulate a special case of the worker/wrapper transformation for changing the result type of a function; because this only saves a few steps in the above calculations, we prefer to use our general formulation.

5 Example: the factorial function

For our second example, we formalise an example from the original article on the worker/wrapper transformation (Peyton Jones & Launchbury, 1991), by showing how to transform a simple definition for the factorial function on integers into a more efficient version that only uses *unboxed* integers.

5.1 Boxed and unboxed integers

Consider the question of how integers should be represented in the implementation of a programming language. For strict languages, such as ML, integers can be represented directly as binary numbers, for example as a 32-bit word. Integers represented in this manner are called *unboxed* integers, because no additional packaging is required around the raw numeric values themselves.

For non-strict languages, such as Haskell, a more refined representation is required, because evaluation is only performed on demand. In particular, we must distinguish between a computation that will return an integer value and an integer value itself. In practice, this is usually achieved using some form of tagging in the implementation. Such tagged integer representations are called *boxed* integers.

Tagging supports the use of non-strict evaluation, but carries a considerable overhead. For example, evaluating $x + y$ conceptually requires evaluating the expressions x and y, unboxing the resulting values (removing the tags), performing the actual addition and then boxing the result (reinstating the tag). It would clearly be more efficient to work with unboxed integers as much as possible.

The key to achieving this behaviour in Haskell is an idea put forward by Peyton Jones and Launchbury (1991), namely to reflect the notion of boxed and unboxed integers directly in the language itself, via the following type definition:

$$\textbf{data}\ Int\ =\ I_{\#}\ Int_{\#}$$

That is, a boxed value of type *Int* can be constructed by applying the constructor tag $I_{\#}$ to a value of type $Int_{\#}$, which is the built-in type of unboxed 32-bit integers, and is unlifted in the sense that it has no $\perp$ element. Note that, by convention in Haskell, unboxed types, values, variables and functions have a # suffix on their names, as in $Int_{\#}$, $0_{\#}$, $n_{\#}$ and $+_{\#}$. However, this symbol has no semantic meaning, and is used purely for identification purposes.

Given the above type definition, converting an unboxed integer into a boxed integer can be achieved simply by applying the constructor $I_{\#}$, and in the opposite direction by removing this constructor through pattern matching. Thus there is no need to define explicit conversion functions.

5.2 Factorial

Consider the following definition for the factorial function on non-negative integers:

$$\begin{aligned} &fac && ::\ Int \rightarrow Int \\ &fac\ 0 && =\ 1 \\ &fac\ n && =\ n * fac\ (n-1) \end{aligned}$$

In a non-strict language such as Haskell, a straightforward implementation of this definition will be rather inefficient, due to repeated unboxing and boxing of integer values. However, because *fac* is strict ($fac\ \perp\ =\ \perp$), it is safe to replace the definition of *fac* by a more efficient *worker* that only uses unboxed integers and strict evaluation, together with a *wrapper* that takes care of the initial unboxing and final reboxing. We show how this transformation can be achieved in practice. Following our recipe, the first step is to redefine *fac* as the least fixed point

```
fac       :: Int → Int
fac       = fix body

body      :: (Int → Int) → (Int → Int)
body f 0  = 1
body f n  = n * f (n − 1)
```

Our aim now is to transform the computation *fac* of type *Int* → *Int* into a worker of type $Int_\#$ → $Int_\#$, as illustrated in the following diagram:

$$\begin{tikzcd}
fac :: \boxed{Int \to Int} \arrow[r, bend left, "unwrap"] & work :: \boxed{Int_\# \to Int_\#} \arrow[l, bend left, "wrap"]
\end{tikzcd}$$

Implementing the conversion functions *wrap* and *unwrap* is simply a matter of taking care of the necessary boxing and unboxing

```
unwrap        :: (Int → Int) → (Int# → Int#)
unwrap f x#   = case f (I# x#) of
                  I# y# → y#

wrap          :: (Int# → Int#) → (Int → Int)
wrap g# x     = case x of
                  I# x# → case g# x# of
                             y# → I# y#
```

Using the above definitions and properties of case expressions, it is straightforward to show that $wrap \circ unwrap = id_{Int \to Int}$, but only if we restrict our attention to argument functions of type *Int* → *Int* that are strict. Hence for this example, the simple assumption $wrap \circ unwrap = id_{Int \to Int}$ is not always valid. However, because *body f* is defined by pattern matching and is therefore strict, we do have the weaker assumption $wrap \circ unwrap \circ body = body$, which is precisely where strictness is exploited in this example. More complex functions than factorial, for which determining strictness depends upon taking account of recursion, will require our weakest worker/wrapper assumption defined using *fix*.

Now that we have satisfied the worker/wrapper preconditions, applying this transformation gives the following result:

```
fac   :: Int → Int
fac   = wrap work

work  :: Int# → Int#
work  = unwrap (body (wrap work))
```

Making the integer argument explicit in the definition for *fac*, and expanding out *wrap*, we obtain the wrapper for the unboxed version of *fac*, which unboxes the integer argument, applies the worker and then boxes the result

$$\begin{aligned}
&\mathit{fac} && :: \mathit{Int} \to \mathit{Int}\\
&\mathit{fac}\ n && = \mathbf{case}\ n\ \mathbf{of}\\
&&& \qquad I_\#\ x_\# \to \mathbf{case}\ \mathit{work}\ x_\#\ \mathbf{of}\\
&&& \qquad\qquad\qquad y_\# \to I_\#\ y_\#
\end{aligned}$$

Now let us simplify the definition for the worker function, which in this case amounts to expanding out definitions and then simplifying the result. As with the *reverse* example, the first step is to redefine *work* using pattern matching, and this transformation can be verified in a similar manner as before

$$\begin{aligned}
\mathit{work}\ x_\# = {} & \mathbf{case}\ (\mathbf{case}\ (I_\#\ x_\#)\ \mathbf{of}\\
& \qquad\qquad 0 \to 1\\
& \qquad\qquad n \to \mathbf{case}\ (n-1)\ \mathbf{of}\\
& \qquad\qquad\qquad I_\#\ a_\# \to \mathbf{case}\ \mathit{work}\ a_\#\ \mathbf{of}\\
& \qquad\qquad\qquad\qquad b_\# \to n * (I_\#\ b_\#))\ \mathbf{of}\\
& \quad I_\#\ y_\# \to y_\#
\end{aligned}$$

Then if we expand out the boxed components 0, 1, n, $-$ and $*$ in terms of the constructor $I_\#$ (for example, by replacing 0 with $I_\#\ 0_\#$), and simplify the resulting definition using properties of case, we obtain a worker with the same structure as the original factorial function, but which now operates entirely using unboxed integers, and hence avoids the need for repeated unboxing and boxing

$$\begin{aligned}
&\mathit{work} && :: \mathit{Int}_\# \to \mathit{Int}_\#\\
&\mathit{work}\ x_\# && = \mathbf{case}\ x_\#\ \mathbf{of}\\
&&& \qquad 0_\# \to 1_\#\\
&&& \qquad n_\# \to n_\#\ *_\#\ \mathit{work}\ (n_\#\ -_\#\ 1_\#)
\end{aligned}$$

Once again, note that once we have made the initial decision regarding the change in type of the function, applying our worker/wrapper recipe is largely a matter of routine equational reasoning, and does not require the use of induction. Note also that this derivation involved changing both the argument and result type of a function, whereas our previous example only changed the result type.

6 Example: the Fibonacci function

For our next example, we show how the worker/wrapper transformation can be used to transform a simple definition for the Fibonacci function on natural numbers into a more efficient version that operates by means of a *memo table*. In contrast to our previous two examples, in which a function is represented by another function, this derivation represents a function by a data structure.

6.1 Memoisation

The term memoisation was coined by Michie (1968), and refers to the optimisation technique of tabulating all the arguments that a function is called with, together with the corresponding results returned, and reusing these results if the function is called again with any of its previous arguments. If this happens often, memoisation can lead to a dramatic improvement in performance.

Our approach to memoising the Fibonacci function is to observe that any function on natural numbers can be represented in an alternative manner as an infinite list (or stream) of results. In particular, we represent a function f as the stream $[f\ 0, f\ 1, f\ 2, \ldots]$ that tabulates its behaviour for all possible argument values.

To implement this representation, let us write *Nat* for the subtype of *Int* comprising the natural numbers $0, 1, 2, 3, \ldots$, and *Stream a* for the subtype of $[a]$ comprising the infinite lists of elements of type a. Using these two subtypes allows us to be more precise about the types of functions that we will define, while still being able to use familiar notation and library functions for integers and lists.

For example, a function *unwrap* that converts functions on natural numbers into streams in the manner described above can now be defined

$$\begin{array}{lcl} \mathit{unwrap} & :: & (\mathit{Nat} \to a) \to \mathit{Stream}\ a \\ \mathit{unwrap}\ f & = & \mathit{map}\ f\ [0\,..] \end{array}$$

Dually, we can also perform conversion in the opposite direction, from a stream to a function, by simply indexing into the stream:

$$\begin{array}{lcl} \mathit{wrap} & :: & \mathit{Stream}\ a \to (\mathit{Nat} \to a) \\ \mathit{wrap}\ xs & = & (xs\ !!) \end{array}$$

The auxiliary operator !! selects the nth element of a stream

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

Using these definitions, we can show that $\mathit{wrap} \circ \mathit{unwrap} = \mathit{id}_{\mathit{Nat} \to a}$ by making the function and numeric arguments explicit, then expanding out definitions to give $(\mathit{unwrap}\ f)\ !!\ n = f\ n$ and finally verifying this equation by induction on n.

The dual property, $\mathit{unwrap} \circ \mathit{wrap} = \mathit{id}_{\mathit{Stream}\ a}$, also holds. In particular, making the stream argument explicit and expanding out definitions gives $\mathit{unwrap}\ (xs\ !!) = xs$, which can be verified by coinduction on streams (Turner, 1995).

In conclusion, the functions *unwrap* and *wrap* establish an isomorphism between the types $\mathit{Nat} \to a$ and *Stream a*. This isomorphism itself is well known, but perhaps surprisingly, generalises in allowing any function type with an inductively defined argument type to be represented in a coinductive manner (Altenkirch, 2001), thereby providing a basis for memoising a large class of functions.

6.2 Fibonacci

Consider the following standard definition for the Fibonacci function on natural numbers, as found in any textbook on numerical series:

```
fib          :: Nat → Nat
fib 0        = 0
fib 1        = 1
fib (n + 2)  = fib n + fib (n + 1)
```

That is, the first two Fibonacci numbers are 0 and 1, and each successive number is the sum of the previous two. While this definition directly expresses the desired behaviour of the function, it also recomputes the same results many times over, in fact requiring exponential time in the size of its argument. We now show how it can be transformed into a more efficient *worker* that constructs a memo table comprising the stream $[0, 1, 1, 2, 3, 5, \ldots]$ of all Fibonacci numbers (and hence ensures that each result is computed at most once), together with a *wrapper* that selects the required element from this table. The first step is to redefine *fib* using *fix*:

```
fib               :: Nat → Nat
fib               = fix body

body              :: (Nat → Nat) → (Nat → Nat)
body f 0          = 0
body f 1          = 1
body f (n + 2)    = f n + f (n + 1)
```

Our aim now is to transform the computation *fib* of type *Nat* → *Nat* into a worker of type *Stream Nat*, as illustrated in the following diagram:

$$\begin{tikzcd}
fib :: \boxed{Nat \to Nat} \arrow[rr, bend left, "unwrap"] & & work :: \boxed{Stream\ Nat} \arrow[ll, bend left, "wrap"]
\end{tikzcd}$$

Because we have already defined appropriate conversion functions satisfying the assumption $wrap \circ unwrap = id_{Nat \to Nat}$ in the previous section, we can now apply the worker/wrapper transformation to obtain the following result:

```
fib   :: Nat → Nat
fib   = wrap work

work  :: Stream Nat
work  = unwrap (body (wrap work))
```

Making the numeric argument explicit in the definition for *fib*, and expanding out *wrap*, we obtain the wrapper for the memoised version of *fib*, which simply selects the required element from the memo table

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

Now we simplify the worker itself, which in this case just amounts to expanding out definitions, and does not require any form of fusion:

$$
\begin{array}{cl}
& \mathit{work}\\
= & \quad \{\ \text{applying } \mathit{work}\ \}\\
& \mathit{unwrap}\ (\mathit{body}\ (\mathit{wrap}\ \mathit{work}))\\
= & \quad \{\ \text{applying } \mathit{unwrap}\ \}\\
& \mathit{map}\ (\mathit{body}\ (\mathit{wrap}\ \mathit{work}))\ [0\,..]\\
= & \quad \{\ \text{applying } \mathit{body}\ \}\\
& \mathit{map}\ (\lambda n \rightarrow \mathbf{case}\ n\ \mathbf{of}\\
& \qquad\qquad\qquad 0 \rightarrow 0\\
& \qquad\qquad\qquad 1 \rightarrow 1\\
& \qquad\qquad\qquad (n+2) \rightarrow \mathit{wrap}\ \mathit{work}\ n + \mathit{wrap}\ \mathit{work}\ (n+1))\ [0\,..]\\
= & \quad \{\ \text{applying } \mathit{wrap}\ \}\\
& \mathit{map}\ (\lambda n \rightarrow \mathbf{case}\ n\ \mathbf{of}\\
& \qquad\qquad\qquad 0 \rightarrow 0\\
& \qquad\qquad\qquad 1 \rightarrow 1\\
& \qquad\qquad\qquad (n+2) \rightarrow \mathit{work}\ !!\ n + \mathit{work}\ !!\ (n+1))\ [0\,..]
\end{array}
$$

That is, we have derived the following definition for the worker, which builds a memo table comprising all Fibonacci numbers, using the table itself to ensure that each number is computed at most once, and lazy evaluation to ensure that the table (which is conceptually infinite) is built on a demand-driven basis

$$
\begin{array}{l}
\mathit{work} :: \mathit{Stream}\ \mathit{Nat}\\
\mathit{work} = \mathit{map}\ f\ [0\,..]\\
\qquad\quad \mathbf{where}\\
\qquad\qquad f\ 0 = 0\\
\qquad\qquad f\ 1 = 1\\
\qquad\qquad f\ (n+2) = \mathit{work}\ !!\ n + \mathit{work}\ !!\ (n+1)
\end{array}
$$

In conclusion, we have improved the time performance of the *fib* function from exponential to quadratic. Further improvements can readily be made in a variety of ways, such as representing the memo table in a more efficient manner (e.g. using an array with constant-time indexing rather than a stream with linear-time indexing), exploiting the structure of the computation to design a specialised version of the memo table (e.g. using a pair of numbers rather than a stream), or using more advanced programming techniques (e.g. cyclic programming). For example, using the technique of unique fixed points (Hinze, 2008), it can be shown that the above definition for the worker is equivalent to the well-known cyclic definition for the stream of Fibonacci numbers

$$
\begin{aligned}
&\mathit{fibs} &&:: \mathit{Stream}\ \mathit{Nat}\\
&\mathit{fibs} &&= 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})
\end{aligned}
$$

7 Example: an evaluation function

For our final example, we show how to use the worker/wrapper transformation in implementing the transformation of a function into *continuation-passing* style, an important precursor to many program analyses and optimisations.

7.1 Continuations

As observed by Reynolds (1972), it is possible to represent any type in an alternative manner using *continuations*. The notion of a continuation can be defined in many ways, but for our purposes it is simply a function that is applied to the result of another computation. Using this idea, we represent a value x as the function $\lambda c \rightarrow c\ x$ that takes another function c (a continuation) as its argument, and applies this function to x in order to produce the final result.

We can implement this representation using the type $(a \rightarrow r) \rightarrow r$ of continuation computations of type a that return results of type r (Wadler, 1992b). For our purposes, however, we only require the special case when the two types are the same, which we abbreviate by $Cont\ a$, and define a function rep that converts normal values into their representation as continuation values:

$$\begin{aligned} &\textbf{type}\ Cont\ a && = (a \rightarrow a) \rightarrow a \\ &rep && :: a \rightarrow Cont\ a \\ &rep\ x && = \lambda c \rightarrow c\ x \end{aligned}$$

Dually, we can also perform conversion in the opposite direction, by supplying the identity function as the continuation:

$$\begin{aligned} &abs && :: Cont\ a \rightarrow a \\ &abs\ f && = f\ id_a \end{aligned}$$

Using these definitions, it is easy to show that $abs \circ rep = id_a$.

7.2 Evaluation

Consider a simple expression language comprising integers values, an addition operator, a single exceptional value called throw and a catch operator:

$$\textbf{data}\ Expr = Val\ Int \mid Add\ Expr\ Expr \mid Throw \mid Catch\ Expr\ Expr$$

This language provides an appropriate minimal setting in which to investigate various aspects of the semantics of exceptions (Hutton & Wright, 2004, 2006, 2007). Informally, *Throw* abandons the current computation and throws an exception, while $Catch\ x\ y$ behaves as the expression x unless it throws an exception, in which case the catch behaves as the handler expression y.

To formalise the meaning of this language, we first recall the *Maybe* type

$$\textbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

That is, a value of type *Maybe a* is either *Nothing*, which we think of as an exceptional value, or has the form *Just x* for some *x* of type *a*, which we think of as a normal value (Spivey, 1990). For the purposes of our example, however, we only require the type *Maybe Int*, which we abbreviate as *Mint*

```
type Mint = Maybe Int
```

Using this type, it is straightforward to define a function that evaluates expressions, and takes care of the necessary propagation and handling of exceptions

```
eval             :: Expr → Mint
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
```

Because of the repeated tagging and untagging of values of type *Mint*, this definition may be rather inefficient. We now show how it can be transformed into a *worker* that uses two continuations to make normal and exceptional control flow explicit (and hence eliminate the use of tags), together with a *wrapper* that takes care of the initial setup. Once again, the first step is to redefine *eval* using *fix*

```
eval               :: Expr → Mint
eval               = fix body

body               :: (Expr → Mint) → (Expr → Mint)
body f (Val n)     = Just n
body f (Add x y)   = case f x of
                       Nothing → Nothing
                       Just n → case f y of
                                  Nothing → Nothing
                                  Just m → Just (n + m)
body f (Throw)     = Nothing
body f (Catch x y) = case f x of
                       Nothing → f y
                       Just n → Just n
```

One might expect now to aim to transform the computation *eval* of type *Expr* → *Mint* into a worker of type *Expr* → *Cont Mint*. In practice, however, the desire to have separate continuations for normal and exceptional control flow, sometimes called *double-barrelled* continuations (Thielecke, 2002), means that we start by splitting the underlying type *Mint* → *Mint* of continuations into two parts, and aim for a worker of the type illustrated in the following diagram:

$$\begin{tikzcd} eval :: \boxed{Expr \to Mint} \arrow[r, bend left, "unwrap"] & work :: \boxed{Expr \to (Int \to Mint) \to Mint \to Mint} \arrow[l, bend left, "wrap"] \end{tikzcd}$$

Intuitively, the worker will apply a success continuation of type $Int \to Mint$ if the result of evaluating the given expression is an integer, or a failure continuation of type *Mint* (which can be viewed as a function with no arguments) if the result is an exception. The necessary conversion functions, satisfying the worker/wrapper assumption $wrap \circ unwrap = id_{Expr \to Mint}$, can be defined as follows:

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

Now that we have established the preconditions for the worker/wrapper transformation, applying this transformation gives

$$\begin{aligned} eval &:: Expr \to Mint \\ eval &= wrap\ work \\ work &:: Expr \to (Int \to Mint) \to Mint \to Mint \\ work &= unwrap\ (body\ (wrap\ work)) \end{aligned}$$

Making the expression argument explicit in the definition for *eval*, and expanding out *wrap*, we obtain the wrapper for the continuation-passing version of *eval*, which simply supplies *Just* and *Nothing* as the initial continuations

$$\begin{aligned} eval &:: Expr \to Mint \\ eval\ e &= work\ e\ Just\ Nothing \end{aligned}$$

Now let us focus our attention on simplifying the definition for the worker function. First of all, we redefine *work* using pattern matching

$$\begin{aligned} work\ (Val\ n)\ s\ f &= s\ n \\ work\ (Add\ x\ y)\ s\ f &= \textbf{case}\ wrap\ work\ x\ \textbf{of} \\ &\qquad Nothing \to f \\ &\qquad Just\ n \to \textbf{case}\ wrap\ work\ y\ \textbf{of} \\ &\qquad\qquad Nothing \to f \\ &\qquad\qquad Just\ m \to s\ (n+m) \\ work\ (Throw)\ s\ f &= f \\ work\ (Catch\ x\ y)\ s\ f &= \textbf{case}\ wrap\ work\ x\ \textbf{of} \\ &\qquad Nothing \to \textbf{case}\ wrap\ work\ y\ \textbf{of} \\ &\qquad\qquad Nothing \to f \\ &\qquad\qquad Just\ n \to s\ n \\ &\qquad Just\ n \to s\ n \end{aligned}$$

Now we simplify the case analyses used in the recursive definition by exposing the implicit use of *unwrap* and then fusing again with *wrap*. In particular, for an arbitrary expression x, and continuations g and s of the appropriate types

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

Applying this result three times gives our final definition for the worker, in which control flow is now managed by explicit success and failure continuations, rather than by repeated tagging and untagging of *Maybe* values

$$\begin{array}{ll}
\mathit{work} & :: \mathit{Expr} \rightarrow (\mathit{Int} \rightarrow \mathit{Mint}) \rightarrow \mathit{Mint} \rightarrow \mathit{Mint} \\
\mathit{work}\ (\mathit{Val}\ n)\ s\ f & = s\ n \\
\mathit{work}\ (\mathit{Add}\ x\ y)\ s\ f & = \mathit{work}\ x\ (\lambda n \rightarrow \mathit{work}\ y\ (\lambda m \rightarrow s\ (n+m))\ f)\ f \\
\mathit{work}\ (\mathit{Throw})\ s\ f & = f \\
\mathit{work}\ (\mathit{Catch}\ x\ y)\ s\ f & = \mathit{work}\ x\ s\ (\mathit{work}\ y\ s\ f)
\end{array}$$

Once again, note that everything follows in a straightforward manner once we have made the initial design decision to represent the type *Mint* using two continuations, and that the derivation does not require the use of induction. We conclude this section by noting that the resulting worker function can now readily be transformed into an abstract machine for evaluating expressions (Ager *et al.*, 2003; Hutton & Wright, 2006), resulting in an efficient machine that operates using two control stacks, one for normal evaluation and the other for handling exceptions.

8 Related work

As discussed in Section 1, the worker/wrapper transformation was first used to exploit strictness information in the Glasgow Haskell Compiler (Peyton Jones & Launchbury, 1991). However, the correctness of the resulting transformation has never been formally verified. During our work on this article we learned that the authors did in fact sketch a proof using the rolling rule, but this proof was never fully developed, nor was it published. The underlying strictness analyser in this compiler has been changed and improved several times since its initial implementation (Peyton Jones & Partain, 1993), but the use of the worker/wrapper technique to exploit this information remains exactly the same.

The worker/wrapper transformation has also been used to improve the shortcut approach to deforestation (Gill *et al.*, 1993), an optimisation technique that attempts to remove intermediate data structures from programs. In particular, the shortcut approach performs fusion when functions that produce and consume lists are written in a specific, idiomatic way, and are completely inlined. In the final chapter of his

PhD thesis (Gill, 1996), Gill describes a scheme for abstracting a function that produces a list in a way that utilised the worker/wrapper transformation to reduce the increase in code size that can result from wholesale inlining of definitions. In particular, Gill considered list producing functions of the form

$$f\ x_1\ \dots\ x_n = \mathit{build}\ (\lambda c\ n \to \texttt{body})$$

which were then split into a wrapper and a worker

$$\begin{aligned} f\ x_1\ \dots\ x_n &= \mathit{build}\ (\lambda c\ n \to \mathit{work}\ x_1\ \dots\ x_n\ c\ n) \\ \mathit{work}\ x_1\ \dots\ x_n\ c\ n &= \texttt{body} \end{aligned}$$

and finally, the new definition for *f* was inlined in the definition for *work*, thereby communicating the deforestation opportunity. However, no formal justification was given for the correctness of the worker/wrapper technique.

Chitil (2000a, 2000b) considerably extended Gill's work, by generalising to allow for multiple return lists, as directed by his type-inference based approach to deforestation. In particular, Chitil uses a static argument transformation (Santos, 1995) to turn a recursive function into a non-recursive function with local recursion, and splits the new, non-recursive function into a worker and wrapper, with detailed justification. He also explores the recursive instance of the worker/wrapper technique as applied to deforestation, acknowledging that 'whereas intuitively the semantic correctness of [this approach] is clear, a formal proof is hard', and suggests that a proof may be possible using improvement theory (Sands, 1998). However, as we have shown in this article, simpler techniques suffice.

Another application of the worker/wrapper transformation has been in changing the order of function arguments, to aid static analysis. In particular, Launchbury and Sheard (1995) use the technique to illustrate how the standard definition

$$\begin{aligned} &\mathit{map} && :: (a \to b) \to [a] \to [b] \\ &\mathit{map}\ f\ \mathit{xs} && = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & && \quad [\,] \to [\,] \\ & && \quad (x : \mathit{xs}) \to f\ x : \mathit{map}\ f\ \mathit{xs} \end{aligned}$$

can be translated into an equivalent version that is defined using a worker that takes the two arguments in the opposite order

$$\begin{aligned} &\mathit{map} && :: (a \to b) \to [a] \to [b] \\ &\mathit{map}\ f\ \mathit{xs} && = \mathit{work}\ \mathit{xs}\ f \\ &\mathit{work} && :: [a] \to (a \to b) \to [b] \\ &\mathit{work}\ \mathit{xs}\ f && = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & && \quad [\,] \to [\,] \\ & && \quad (x : \mathit{xs}) \to f\ x : \mathit{work}\ \mathit{xs}\ f \end{aligned}$$

It is interesting to note that the various users of the worker/wrapper transformation appear to realise its potential as a powerful tool for communicating a change of type through a recursive definition, but did not explore this generalisation.

The worker/wrapper terminology has also been used to describe a similar but distinct technique, namely when a wrapper of one type is exposed to an optimisation system for the purpose of applying a type conversion to a worker. Technically, this corresponds to the *adapter* design pattern (Gamma *et al.*, 1995). The rewrite system of the Glasgow Haskell Compiler (Peyton Jones *et al.*, 2001) uses this technique in its implementation of short-cut deforestation, as does the recent implementation of stream fusion (Coutts *et al.*, 2007). In both these systems, rather than wholesale inlining of a candidate function, an inlined wrapper expresses the opportunity for deforestation, but the actual computation is performed by the worker. In a real sense, these uses of workers and wrappers are part of the worker/wrapper *transformation*, which is the combination of *splitting* a function into a worker and wrapper in a correct and potentially useful manner, together with the *use* of these new functions to fulfill specific roles inside a rewrite system.

9 Conclusion and further work

In this article we presented the worker/wrapper transformation, and gave several examples of its use as an equational reasoning technique for improving the performance of functional programs. In particular, we showed how it can be used to derive more efficient programs based upon the use of accumulation, unboxed values, memoisation and continuations. Since these are all well-studied and widely used approaches to program optimisation, it is natural to ask what new value the worker/wrapper transformation brings? We see the following three primary benefits:

- It provides a general and systematic approach to transforming a computation of one type into an equivalent computation of another type. Such type transformations are pervasive in functional programming and reasoning, and in the efficient compilation of functional languages.
- It is straightforward to understand and apply, requiring only basic equational reasoning techniques, and often avoiding the need for induction. As such, it can readily be utilised by a wide spectrum of functional programmers as a simple but powerful technique for refactoring their programs.
- It captures many seemingly unrelated optimisation techniques inside a single unified framework, which may help reveal new connections between existing techniques, and the discovery of new techniques.

This article is by no means the end of the worker/wrapper story, but rather opens up a number of opportunities for further research. Interesting topics for further work include mechanising the technique in an equational reasoning system such as HERA (Gill, 2006), investigating how it can be automated in an optimising compiler such as GHC (Peyton Jones *et al.*, 2001), considering programs that utilise effects (Wadler, 1992a; McBride & Paterson, 2008) and more advanced applications (e.g. to programming languages themselves), exploring the use of specialised patterns of recursion (Gibbons & Jones, 1998; Hutton, 1999) and versions of the transformation (e.g. for changing the result type of a function) and further generalising the technique using category theory (Backhouse *et al.*, 1995).

Acknowledgments

We would like to thank Roland Backhouse, Olaf Chitil, Isaac Dupree, Conal Elliott, Peter Gammie, Peter Jonsson, John Matthews, Conor McBride, Simon Peyton Jones and Mark Tullsen for many useful comments and suggestions, Colin Runciman for the invitation to submit to JFP, and the three anonymous referees for their detailed reviews. Work on this article was initiated during a sabbatical visit to Galois, Inc. in Portland, Oregon by the second author during Summer 2007, for which funding from Galois is very gratefully acknowledged.

References

Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier & Midtgaard, Jan (2003) A functional correspondence between evaluators and abstract machines. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* Uppsala, Sweden.

Altenkirch, Thorsten (2001) Representations of first-order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications*. LNCS, no. 2044. Berlin: Springer.

Backhouse, Roland (2002) Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. LNCS Tutorial, vol. 2297. Berlin: Springer-Verlag.

Backhouse, Roland, Bijsterveld, Marcel, van Geldrop, Rik & van der Woude, Jaap (1995) Categorical fixed point calculus. In *Proceedings of the Sixth International Conference on Category Theory and Computer Science*. Berlin: Springer-Verlag.

Bird, Richard (1998) *Introduction to Functional Programming using Haskell*, second edition. New York: Prentice Hall.

Burstall, Rod & Darlington, John (1977) A transformational system for developing recursive programs. *J. ACM* **24**, 44–67.

Chitil, Olaf (2000a) *Type-Inference Based Deforestation of Functional Programs*. Ph.D. thesis, RWTH Aachen.

Chitil, Olaf (2000b) Type-inference based short cut deforestation (nearly) without inlining. In *Proceedings of 11th International Workshop on Implementation of Functional Languages*, Chris Clack and Pieter Koopman (eds). LNCS, no. 1868. Berlin: Springer.

Coutts, Duncan, Leshchinskiy, Roman & Stewart, Don (2007) Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*. New York: ACM Press.

Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John (1995) *Design Patterns*. Reading, Mass: Addison-Wesley Professional.

Gibbons, Jeremy & Jones, Geraint (1998). The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* Baltimore, Maryland.

Gill, Andy (1996) *Cheap Deforestation for Non-Strict Functional Languages*. Ph.D. thesis, University of Glasgow.

Gill, Andy (2006) Introducing the haskell equational reasoning assistant. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. New York: ACM Press.

Gill, Andy, Launchbury, John & Peyton Jones, Simon (1993) A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. New York: ACM Press.

Hinze, Ralf (2008) Functional pearl: Streams and unique fixed points. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* Victoria, British Columbia.

Hoare, Tony (1972) Proof of correctness of data representations. *Acta Informatica* **1**(4), 271–281.

Hughes, John (1986) A novel representation of lists and its application to the function reverse. *Info. Process. Lett.* **22**(3), pp 141–144.

Hutton, Graham (1999) A tutorial on the universality and expressiveness of fold. *J. Funct. Prog.* **9**(4), pp 355–372.

Hutton, Graham (2007) *Programming in Haskell*, Cambridge: Cambridge University Press.

Hutton, Graham & Wright, Joel (2004) Compiling exceptions correctly. In *Proceedings of the Seventh International Conference on Mathematics of Program Construction.* LNCS, vol. 3125. Stirling, Scotland: Springer.

Hutton, Graham & Wright, Joel (2006) Calculating an exceptional machine. Selected papers from the *Fifth Symposium on Trends in Functional Programming*, Munich, vol. 5, November 2004 UK: Intellect.

Hutton, Graham & Wright, Joel (2007) What is the meaning of these constant interruptions? *J. Funct. Prog.* **17**(6), pp 777–792.

Launchbury, John & Sheard, Tim (1995) Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the Seventh ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture.* New York: ACM Press.

McBride, Conor & Paterson, Ross (2008) Applicative programming with effects. *J. Funct. Prog.* **18**(1), pp 1–13.

Meijer, Erik, Fokkinga, Maarten & Paterson, Ross (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, Hughes, John (ed). LNCS, no. 523. Springer-Verlag.

Michie, Donald (1968) Memo functions and machine learning. *Nature*, **218**, pp 19–22.

Peyton Jones, Simon (2003) *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press. Also available on `www.haskell.org/definition`.

Peyton Jones, Simon & Launchbury, John (1991) Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the Conference on Functional Programming and Computer Architecture.* Cambridge, MA: Springer-Verlag.

Peyton Jones, Simon & Partain, Will (1993) Measuring the effectiveness of a simple strictness analyser. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, Hammond, Kevin & O'Donnell, John (eds). Ayr, Scotland: Springer-Verlag.

Peyton Jones, Simon, Tolmach, Andrew & Hoare, Tony (2001) Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Haskell.* ACM Press.

Reynolds, John C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference.* ACM Press.

Sands, David (1998) Improvement theory and its applications. In *Higher Order Operational Techniques in Semantics*, Gordon, Andrew & Pitts, Andrew (eds). Cambridge, UK: Cambridge University Press.

Santos, Andre (1995) *Compilation by Transformation in Non-strict Functional Languages.* Ph.D. thesis, University of Glasgow.

Schmidt, David (1986) *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc.

Spivey, Mike (1990) A functional theory of exceptions. *Sci. Comput. Prog.* **14**(1), pp 25–42.

Thielecke, Hayo (2002) Comparing control constructs by double-barrelled CPS. *Higher-Order Symbol. Comput.* **15**(2/3), pp 141–160.

Turner, David A. (1995) Elementary strong functional programming. In *Proceedings of the First International Symposium on Functional Programming Languages in Education.* LNCS, no. 1022. Springer-Verlag.

Wadler, Philip (1992a) Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, Broy, Manfred (ed). Springer–Verlag.

Wadler, Philip (1992b) The essence of functional programming. *Proc. Principles Prog. Lang.* New York: ACM Press, pp 1–14.

Appendix: fixed points and the rolling rule

For completeness, this appendix briefly reviews the necessary semantic theory of fixed points that underlies our formalisation of the worker/wrapper transformation, and presents a proof of the rolling rule in this context.

Our formalisation is based upon the domain-theoretic approach to semantics (Schmidt, 1986), in which the basic idea is that types are *complete partial orders* (cpos), that is, sets with a partial-ordering $\sqsubseteq$, the least element $\bot$, and limits of all non-empty chains. In turn, programs are *continuous functions*, that is, functions between cpos that preserve the partial-order and limit structure.

Now consider a recursive equation $x = f\ x$ that defines a value x in terms of itself and some continuous function f. A well-known fixed point theorem states that this equation has the least solution for x, denoted by $\mathit{fix}\ f$ and called the *least fixed point* of f, which is adopted as the semantics of the definition. Moreover, $\mathit{fix}\ f$ is constructed as the limit of the following infinite chain:

$$\bot \ \sqsubseteq\ f\ \bot \ \sqsubseteq\ f\ (f\ \bot)\ \sqsubseteq\ f\ (f\ (f\ \bot))\ \sqsubseteq\ \cdots$$

As a simple example of this approach, consider the equation $\mathit{ones} = 1 : \mathit{ones}$ that defines the infinite list $1 : 1 : 1 : \cdots$. This definition can be rewritten as $\mathit{ones} = f\ \mathit{ones}$, where f is the function defined by $f\ \mathit{xs}\ =\ 1 : \mathit{xs}$. Hence, the semantics of the definition is given by $\mathit{ones} = \mathit{fix}\ f$, and by the fixed point theorem is constructed as the limit of the infinite chain of partial lists containing increasing numbers of ones

$$\bot\ \sqsubseteq\ 1 : \bot\ \sqsubseteq\ 1 : 1 : \bot\ \sqsubseteq\ 1 : 1 : 1 : \bot\ \sqsubseteq\ \cdots$$

The fixed point approach to recursive definitions can be realised in Haskell by defining an explicit version of the function fix as follows:

$$\begin{aligned} &\mathit{fix} && :: (a \to a) \to a \\ &\mathit{fix}\ f && = f\ (\mathit{fix}\ f) \end{aligned}$$

For example, evaluating the expression $\mathit{fix}\ (\lambda xs \to 1 : xs)$ using this definition gives the infinite list of ones, $1 : 1 : 1 : \cdots$, as expected.

In order to prove the rolling rule, which states that $\mathit{fix}\ (g \circ f) = g\ (\mathit{fix}\ (f \circ g))$ for any functions f and g of the appropriate types, we exploit the following two fundamental properties of fixed points (Backhouse, 2002):

$$\begin{aligned} &\mathit{fix}\ f\ = f\ (\mathit{fix}\ f) && \text{(computation)} \\ &f\ x \sqsubseteq x\ \Rightarrow\ \mathit{fix}\ f \sqsubseteq x && \text{(induction)} \end{aligned}$$

The first property states that *fix f* is a fixed point of *f* (an expression x satisfying $f\ x = x$), while the second states that *fix f* is the least *prefix* point of *f* (the least expression x satisfying $f\ x \sqsubseteq x$). One might have expected in the latter case to state that *fix f* is the least *fixed* point rather than just the least prefixed point, but these two notions can be shown to be equivalent, and the above formulation has the advantage of being more useful for reasoning purposes. Using these two properties, the rolling rule can now be verified by mutual inclusion

$$\begin{array}{ll}
 & \mathit{fix}\ (g \circ f) \sqsubseteq g\ (\mathit{fix}\ (f \circ g)) \\
\Leftarrow & \quad \{\ \text{induction}\ \} \\
 & (g \circ f)\ (g\ (\mathit{fix}\ (f \circ g))) \sqsubseteq g\ (\mathit{fix}\ (f \circ g)) \\
\Leftrightarrow & \quad \{\ \text{applying} \circ\ \} \\
 & g\ (f\ (g\ (\mathit{fix}\ (f \circ g)))) \sqsubseteq g\ (\mathit{fix}\ (f \circ g)) \\
\Leftrightarrow & \quad \{\ \text{unapplying} \circ\ \} \\
 & g\ ((f \circ g)\ (\mathit{fix}\ (f \circ g))) \sqsubseteq g\ (\mathit{fix}\ (f \circ g)) \\
\Leftrightarrow & \quad \{\ \text{computation}\ \} \\
 & g\ (\mathit{fix}\ (f \circ g)) \sqsubseteq g\ (\mathit{fix}\ (f \circ g)) \\
\Leftrightarrow & \quad \{\ \text{reflexivity}\ \} \\
 & \mathit{True}
\end{array}$$

and

$$\begin{array}{ll}
 & g\ (\mathit{fix}\ (f \circ g)) \sqsubseteq \mathit{fix}\ (g \circ f) \\
\Leftrightarrow & \quad \{\ \text{computation}\ \} \\
 & g\ (\mathit{fix}\ (f \circ g)) \sqsubseteq (g \circ f)\ (\mathit{fix}\ (g \circ f)) \\
\Leftrightarrow & \quad \{\ \text{applying} \circ\ \} \\
 & g\ (\mathit{fix}\ (f \circ g)) \sqsubseteq g\ (f\ (\mathit{fix}\ (g \circ f))) \\
\Leftarrow & \quad \{\ g \text{ is monotonic}\ \} \\
 & \mathit{fix}\ (f \circ g) \sqsubseteq f\ (\mathit{fix}\ (g \circ f)) \\
\Leftrightarrow & \quad \{\ \text{above result, swapping } f \text{ and } g\ \} \\
 & \mathit{True}
\end{array}$$

We conclude by noting that the above proof of the rolling rule only relies on the basic notion of monotonic functions on partially ordered sets, rather than the stronger notion of continuous functions on cpos. However, working in this stronger setting automatically guarantees that the necessary fixed points always exist, by virtue of the fixed point theorem described above, which in weaker settings may become a side condition on the rolling rule.