Factorising Folds for Faster Functions (Extended Version)

GRAHAM HUTTON
University of Nottingham, UK

MAURO JASKELIOFF
Universidad Nacional de Rosario, Argentina

ANDY GILL
University of Kansas, USA

Abstract

The worker/wrapper transformation is a general technique for improving the performance of recursive programs by changing their types. The previous formalisation (Gill & Hutton, 2009) was based upon a simple fixed point semantics of recursion. In this article we develop a more structured approach, based upon initial algebra semantics. In particular, we show how the worker/wrapper transformation can be applied to programs defined using the structured pattern of recursion captured by fold operators, and illustrate our new technique with a number of examples.

1 Introduction

The worker/wrapper transformation is a general technique for changing the type of a recursive program to improve its performance. The basic idea is simple and pervasive: given a recursive program of some type, we aim to factorise it into a more efficient *worker* program of a different type, together with a *wrapper* program that acts as an interface between the original program and the new worker.

Special cases of the worker/wrapper transformation have been utilised for many years, particularly in optimizing compilers. For example, the technique has been used in the Glasgow Haskell Compiler since its inception, to replace the use of boxed data structures by more efficient unboxed data structures when safe to do so (Peyton Jones & Launchbury, 1991). However, it is only recently that the technique has been formalised, proved correct, and presented as a general approach to improving the performance of programs by improving the choice of data structures (Gill & Hutton, 2009).

The previous formalisation of the technique was based upon a simple fixed point semantics of recursive programs. In this article we develop a more structured approach, based upon initial algebra semantics. In particular, we show how the worker/wrapper transformation can be applied to programs defined using the structured pattern of recursion captured by fold operators. More precisely, the article makes the following contributions:

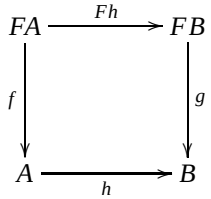
- We show how the worker/wrapper transformation applies to programs defined using folds, by generalising to a categorical view of types as initial algebras.
- We identify four conditions for the correctness of the transformation, and show that these conditions form a simple lattice structure.
- We illustrate our technique with a number of examples, including a correctness proof for a new approach to implementing substitution efficiently (Voigtländer, 2008).

The article is aimed at readers who are familiar with the basics of initial algebra semantics, say to the level of chapter two of Bird & de Moor (1997), but no previous experience with the worker/wrapper transformation is assumed.

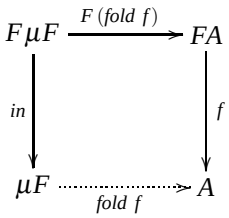
2 Initial algebra semantics

The recursion operator *fold* encapsulates a common pattern for defining functions that consume values of a recursively defined type (Hutton, 1999). In this section we review the categorical treatment of *fold* in terms of initial algebras, and introduce our notation. For further details, see (Malcolm, 1990; Meijer *et al.*, 1991; Bird & de Moor, 1997).

Suppose that we fix a category $\mathbf{C}$ and a functor $F : \mathbf{C} \to \mathbf{C}$ on this category. Then the notion of an *algebra* is defined as a pair (A, f) comprising an object A and an arrow $f : FA \to A$, and a *homomorphism* $h : (A, f) \to (B, g)$ from one such algebra to another is an arrow $h : A \to B$ such that the following diagram commutes:



Algebras and homomorphisms themselves form a category, with composition and identities inherited from $\mathbf{C}$. An *initial algebra* is an initial object in this new category, and we write $(\mu F, in)$ for an initial algebra, and *fold* f for the unique homomorphism $h : (\mu F, in) \to (A, f)$ from the initial algebra to any other algebra (A, f). That is, *fold* f is defined as the unique arrow that makes the following diagram commute:



In the literature, *fold* f is sometimes written using the banana brackets notation $(\!|f|\!)$, and termed a *catamorphism*. The above definition for *fold* f can also be expressed as the following equivalence, known as the *universal property* of *fold*:

$$h = \mathit{fold}\ f \quad \Leftrightarrow \quad h \circ \mathit{in} = f \circ Fh$$

The $\Rightarrow$ direction states that $fold\ f$ is a homomorphism from the initial algebra $(\mu F, in)$ to another algebra (A, f), while the $\Leftarrow$ direction states that any other such homomorphism h must be equal to $fold\ f$. Taken as a whole, the universal property expresses in an equational manner the fact that $fold\ f$ is the unique homomorphism from $(\mu F, in)$ to (A, f).

The universal property can be used to verify the well-known *fusion* property of *fold*, which states that the composition of a function and a *fold* can always be re-expressed as a single *fold*, provided the function is a homomorphism of the appropriate type:

$$\frac{h \circ f \ = \ g \circ Fh}{h \circ fold\ f \ = \ fold\, g}$$

Proof:

$$\begin{array}{ll} & h \circ fold\ f \ = \ fold\, g \\ \Leftrightarrow & \quad \{ \text{ universal property of } fold \ \} \\ & h \circ fold\ f \circ in \ = \ g \circ F\,(h \circ fold\ f) \\ \Leftrightarrow & \quad \{ \ F \text{ is a functor } \} \\ & h \circ fold\ f \circ in \ = \ g \circ Fh \circ F\,(fold\ f) \\ \Leftrightarrow & \quad \{ \ fold\ f \text{ is a homomorphism } \} \\ & h \circ f \circ F\,(fold\ f) \ = \ g \circ Fh \circ F\,(fold\ f) \\ \Leftarrow & \quad \{ \text{ extensionality } \} \\ & h \circ f \ = \ g \circ Fh \end{array}$$

□

2.1 Example: finite lists

Suppose that we define a functor L on the category **SET** by $L\,A = 1 + (\mathbb{Z} \times A)$, where $\mathbb{Z}$ is the integers. Then an algebra is a pair (A, f) comprising a set A and a function $f : 1 + (\mathbb{Z} \times A) \to A$. Functions of this type can always be uniquely decomposed into the form $f = [g, h]$ for some $g : 1 \to A$ and $h : \mathbb{Z} \times A \to A$, and a homomorphism $k : (A, [g, h]) \to (B, [i, j])$ from one such algebra to another is given by a function $k : A \to B$ such that

$$\begin{array}{lcl} k \circ g & = & i \\ k \circ h & = & j \circ (id_{\mathbb{Z}} \times k) \end{array}$$

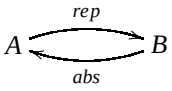
The functor L has an initial algebra $(\mu L, in) = (List(\mathbb{Z}), [nil, cons])$, where $List(\mathbb{Z})$ is the set of finite lists of integers, and $nil : 1 \to List(\mathbb{Z})$ and $cons : \mathbb{Z} \times List(\mathbb{Z}) \to List(\mathbb{Z})$ are constructors for this set. Given any other set A and functions $i : 1 \to A$ and $j : \mathbb{Z} \times A \to A$, the function $fold\ [i, j] : List(\mathbb{Z}) \to A$ is uniquely defined by:

$$\begin{array}{lcl} fold\ [i, j] \circ nil & = & i \\ fold\ [i, j] \circ cons & = & j \circ (id_{\mathbb{Z}} \times fold\ [i, j]) \end{array}$$

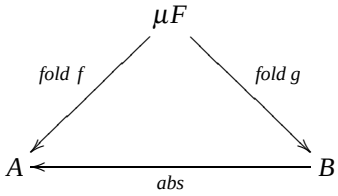
That is, $fold\ [i, j]$ replaces the *nil* constructor at the end of a list by the function i, and each *cons* constructor within the list by the function j. For example, the function $sum : List(\mathbb{Z}) \to \mathbb{Z}$ that sums a list of integers can be defined by $sum = fold\ [zero, plus]$, where $zero : 1 \to \mathbb{Z}$ and $plus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ are given by $zero\ () = 0$ and $plus\ (x, y) = x + y$.

3 Worker/wrapper for folds

Consider the problem of changing the return type of a *fold* to improve its performance. More precisely, suppose we are given a function $fold\ f : \mu F \to A$ for some $f : FA \to A$, and we wish to change the return type from A to some other type B. The worker/wrapper approach to this problem is based upon the use of conversion functions



with the property that $abs \circ rep = id_A$. 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 in the terminology of data representation (Hoare, 1972) expresses that the *abstract* type A can be faithfully represented by the *concrete* type B. Given such a setting, we now seek conditions under which the following diagram commutes:



That is, in worker/wrapper terminology, we seek conditions that allow the original recursive function $fold\ f$ that produces a result of type A to be factorised as the composition of a recursive *worker* function $fold\ g$ that produces a result of type B, and a *wrapper* function *abs* that converts the result back to the original type A.

One approach to solving this problem is to simply apply fusion. Even though this property is normally viewed as being concerned with combining a function with a *fold*, it can also be viewed in the opposite direction as providing a sufficient condition for the factorisation, or *fission* (Gibbons, 2006), of a *fold*:

$$\begin{array}{ll} & fold\ f\ =\ abs \circ fold\ g \\ \Leftarrow & \quad \{\ \text{fusion property of } fold\ \} \\ & f \circ F\ abs\ =\ abs \circ g \end{array}$$

However, given the assumption that $abs \circ rep = id_A$, we can in fact identify four relevant conditions, of which the equation arising from fusion is just one instance:

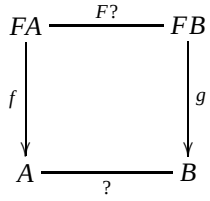
$$(1) \quad g\ =\ rep \circ f \circ F\ abs$$

$$(2) \quad rep \circ f\ =\ g \circ F\ rep$$

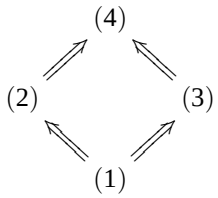
$$(3) \quad f \circ F\ abs\ =\ abs \circ g$$

$$(4) \quad f\ =\ abs \circ g \circ F\ rep$$

These equations correspond to the four possible ways of completing the following diagram by replacing each question mark with either *rep* or *abs*:



What do these conditions express, and how do they relate? Equation (1) provides an explicit definition for g in terms of f; (2) states that *rep* is a homomorphism from f to g; (3) states that *abs* is a homomorphism from g to f (the condition that directly arises from the use of fusion); and (4) provides a definition for f in terms of g. Together, they form a simple lattice, with (1) as the strongest condition and (4) as the weakest:



Proof: (1) $\Rightarrow$ (2)

$$
\begin{array}{ll}
 & g \circ F\,rep \;=\; rep \circ f \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
 & g \circ F\,rep \;=\; rep \circ f \circ id_{FA} \\
\Leftrightarrow & \quad \{ F \text{ is a functor } \} \\
 & g \circ F\,rep \;=\; rep \circ f \circ F\,id_A \\
\Leftrightarrow & \quad \{ abs \circ rep = id_A \} \\
 & g \circ F\,rep \;=\; rep \circ f \circ F\,(abs \circ rep) \\
\Leftrightarrow & \quad \{ F \text{ is a functor } \} \\
 & g \circ F\,rep \;=\; rep \circ f \circ F\,abs \circ F\,rep \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & g \;=\; rep \circ f \circ F\,abs
\end{array}
$$

□

Proof: (1) $\Rightarrow$ (3)

$$
\begin{array}{ll}
 & abs \circ g \;=\; f \circ F\,abs \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
 & abs \circ g \;=\; id_A \circ f \circ F\,abs \\
\Leftrightarrow & \quad \{ abs \circ rep = id_A \} \\
 & abs \circ g \;=\; abs \circ rep \circ f \circ F\,abs \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & g \;=\; rep \circ f \circ F\,abs
\end{array}
$$

□

Proof: (2) ⇒ (4)

$$
\begin{array}{ll}
 & f \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
 & id_A \circ f \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \; abs \circ rep = id_A \; \} \\
 & abs \circ rep \circ f \;=\; abs \circ g \circ F\, rep \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & rep \circ f \;=\; g \circ F\, rep
\end{array}
$$

□

Proof: (3) ⇒ (4)

$$
\begin{array}{ll}
 & f \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
 & f \circ id_{FA} \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \; F \text{ is a functor } \} \\
 & f \circ F\, id_A \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \; abs \circ rep = id_A \; \} \\
 & f \circ F\, (abs \circ rep) \;=\; abs \circ g \circ F\, rep \\
\Leftrightarrow & \quad \{ \; F \text{ is a functor } \} \\
 & f \circ F\, abs \circ F\, rep \;=\; abs \circ g \circ F\, rep \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & f \circ F\, abs \;=\; abs \circ g
\end{array}
$$

□

It is now straightforward to verify that each of the first three conditions implies the desired factorisation result, namely that $fold\ f = abs \circ fold\ g$. In particular, we already know that (1) implies (3) using the lattice diagram, and that (3) implies the desired result using fusion, hence it only remains to verify that condition (2) is also sufficient:

$$
\begin{array}{ll}
 & fold\ f \;=\; abs \circ fold\ g \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
 & id_A \circ fold\ f \;=\; abs \circ fold\ g \\
\Leftrightarrow & \quad \{ \; abs \circ rep = id_A \; \} \\
 & abs \circ rep \circ fold\ f \;=\; abs \circ fold\ g \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & rep \circ fold\ f \;=\; fold\ g \\
\Leftarrow & \quad \{ \text{ fusion property of } fold \; \} \\
 & rep \circ f \;=\; g \circ F\, rep
\end{array}
$$

□

The situation regarding (4) is more complicated, and we will return to this shortly. In the meantime, let us consider how the first three conditions are used in practice.

For some applications, the definition for the function g that forms the body of the worker $fold\ g$ will already be given, and our aim then is to *verify* that one of the three conditions is satisfied, to ensure that the worker/wrapper factorisation holds. For many applications,

however, our aim will be to *construct* a suitable function g. In such cases, condition (1) provides an explicit definition $g = rep \circ f \circ F\ abs$ for the body of the worker in a similar manner to (Gill & Hutton, 2009), and our aim then is to *simplify* the definition. This simplification process is typically driven by the desire to fuse together instances of *rep* and *abs*, to eliminate the overhead of repeatedly converting between the concrete and abstract types. In contrast, conditions (2) and (3) provide a specification for g, and our aim is then to *calculate* a definition that satisfies the specification, again with the desire to fuse together instances of the conversion functions between the two types.

Given that (1) is the strongest condition and provides an explicit definition for g as a starting point, why would we ever wish to use the other conditions? In our experience, using one of the weaker conditions often results in a simpler verification or calculation process. In combination with the fact that (3) corresponds to the familiar case of fusion, for the remainder of the article we will primarily focus on (2). Nonetheless, it is interesting to consider the other conditions, and their relationships.

3.1 The weakest condition

Let us now return to the remaining condition in our lattice:

$$(4) \quad f \ = \ abs \circ g \circ F\ rep$$

Unfortunately, in general this condition does not imply that $fold\ f = abs \circ fold\ g$, and is only sufficient to ensure the following more specialised worker/wrapper factorisation in which the body g of the worker is composed with an additional term:

$$fold\ f \ = \ abs \circ fold\ (g \circ F\ (rep \circ abs))$$

Proof:

$$\begin{array}{ll} & fold\ f \ = \ abs \circ fold\ (g \circ F\ (rep \circ abs)) \\ \Leftarrow & \quad \{ \text{ fusion property of } fold \ \} \\ & abs \circ g \circ F\ (rep \circ abs) \ = \ f \circ F\ abs \\ \Leftrightarrow & \quad \{ \ F \text{ is a functor } \} \\ & abs \circ g \circ F\ rep \circ F\ abs \ = \ f \circ F\ abs \\ \Leftarrow & \quad \{ \text{ extensionality } \} \\ & abs \circ g \circ F\ rep \ = \ f \end{array}$$

□

The additional term $F\ (rep \circ abs)$ in the worker plays the role of a *normalisation* function that is applied after each recursive call. In general, $rep \circ abs \neq id_B$, but we can think of $rep \circ abs$ as normalising a value of type B by first converting to the type A, which is typically a 'smaller' type, and then converting back to B.

It is natural to ask when (4) does imply $fold\ f = abs \circ fold\ g$. The answer is given by the following condition, which states that $rep \circ abs$ is a homomorphism from g to itself:

$$(5) \quad rep \circ abs \circ g \ = \ g \circ F\ (rep \circ abs)$$

In particular, we then have the following equivalence:

$$(4) \wedge (5) \quad \Leftrightarrow \quad (2) \wedge (3)$$

That is, the combination of (4) and (5) is equivalent to the combination of (2) and (3), either condition of which we have already shown implies the worker/wrapper factorisation. To verify the $\Rightarrow$ direction, we first show that $(4) \wedge (5) \Rightarrow (2)$:

$$
\begin{array}{ll}
 & rep \circ f \\
= & \quad \{ \ (4) \ \} \\
 & rep \circ abs \circ g \circ F\ rep \\
= & \quad \{ \ (5) \ \} \\
 & g \circ F\ (rep \circ abs) \circ F\ rep \\
= & \quad \{ \ F \text{ is a functor} \ \} \\
 & g \circ F\ (rep \circ abs \circ rep) \\
= & \quad \{ \ abs \circ rep = id_A \ \} \\
 & g \circ F\ rep
\end{array}
$$

□

And similarly for $(4) \wedge (5) \Rightarrow (3)$:

$$
\begin{array}{ll}
 & f \circ F\ abs \\
= & \quad \{ \ (4) \ \} \\
 & abs \circ g \circ F\ rep \circ F\ abs \\
= & \quad \{ \ F \text{ is a functor} \ \} \\
 & abs \circ g \circ F\ (rep \circ abs) \\
= & \quad \{ \ (5) \ \} \\
 & abs \circ rep \circ abs \circ g \\
= & \quad \{ \ abs \circ rep = id_A \ \} \\
 & abs \circ g
\end{array}
$$

□

For the $\Leftarrow$ direction, we have already shown that $(2) \Rightarrow (4)$ and $(3) \Rightarrow (4)$, so all that remains to verify is $(2) \wedge (3) \Rightarrow (5)$, which proceeds as follows:

$$
\begin{array}{ll}
 & g \circ F\ (rep \circ abs) \\
= & \quad \{ \ F \text{ is a functor} \ \} \\
 & g \circ F\ rep \circ F\ abs \\
= & \quad \{ \ (2) \ \} \\
 & rep \circ f \circ F\ abs \\
= & \quad \{ \ (3) \ \} \\
 & rep \circ abs \circ g
\end{array}
$$

□

We conclude this section by noting that condition (5) also implies the following property, which is precisely the worker/wrapper fusion property from (Gill & Hutton, 2009) for the special case when the worker is defined using *fold*:

$$(6) \quad rep \circ abs \circ fold\ g \ = \ fold\ g$$

That is, even though the identity $rep \circ abs = id_B$ does not always hold, given (5) this identity does hold for the special case of values of type B that are produced by the worker itself. The proof of worker/wrapper fusion is now a simple application of fusion:

$$
\begin{array}{ll}
 & rep \circ abs \circ fold\ g\ =\ fold\ g \\
\Leftarrow & \quad \{\ \text{fusion property of } fold\ \} \\
 & rep \circ abs \circ g\ =\ g \circ F\ (rep \circ abs)
\end{array}
$$

□

4 Worker/wrapper for lists

To illustrate our new worker/wrapper technique, we now move from the abstract world of category theory to the concrete world of Haskell (Peyton Jones, 2003). Our first example concerns lists, for which the *fold* operator in Haskell is defined as follows:

$$
\begin{array}{lcl}
fold & :: & (a \to b \to b) \to b \to [a] \to b \\
fold\ f\ v\ [\,] & = & v \\
fold\ f\ v\ (x : xs) & = & f\ x\ (fold\ f\ v\ xs)
\end{array}
$$

That is, the function $fold\ f\ v$ processes a list by replacing the empty list $[\,]$ by the value v, and each constructor $(:)$ within the list by the function f. For example, the function that sums a list of numbers can be defined by $sum = fold\ (+)\ 0$.

Now suppose we are given a function $fold\ f\ v :: [a] \to b$ for some $f :: a \to b \to b$ and $v :: b$, and that we wish to change the return type of the *fold* from b to some other type c. Moreover, we also assume that we are given conversion functions $rep :: b \to c$ and $abs :: c \to b$ satisfying the equation $abs \circ rep = id_b$. Then instantiating our general theory from the previous section, we find that any of the three conditions

$$(1) \quad g\ x\ y = rep\ (f\ x\ (abs\ y))$$

$$(2) \quad rep\ (f\ x\ y) = g\ x\ (rep\ y)$$

$$(3) \quad f\ x\ (abs\ y) = abs\ (g\ x\ y)$$

is sufficient to justify the following factorisation of the original *fold* that produces a result of type b into the composition of a worker *fold* that produces a result of type c, and a wrapper function that converts the result back to the original type b:

$$fold\ f\ v \quad = \quad abs \circ fold\ g\ (rep\ v)$$

4.1 Example: fast reverse

Consider the problem of transforming a simple function that reverses a list into a more efficient version that uses accumulation. This transformation is normally achieved using more elementary techniques (Hutton, 2007), but we now show that it also fits naturally into our worker/wrapper paradigm based upon *fold*, and leads to a simpler derivation than the previous worker/wrapper approach based upon *fix*.

Using explicit recursion, a reverse function can be defined by

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

or equivalently, using the *fold* operator for lists:

$$\begin{array}{lcl} \mathit{rev} & :: & [a] \to [a] \\ \mathit{rev} & = & \mathit{fold}\ \mathit{snoc}\ [\,] \\ \mathit{snoc} & :: & a \to [a] \to [a] \\ \mathit{snoc}\ x\ xs & = & xs +\!\!+ [x] \end{array}$$

However, because of the use of append ($+\!\!+$), this definition for *rev* takes quadratic time. We now show how our worker/wrapper technique for *fold* can be used to derive a more efficient worker that uses an extra argument to accumulate the result, together with a wrapper that takes care of the initial setup. Using the notion of currying, the introduction of an accumulator argument corresponds to changing the return type of *rev* from a list to a function on lists, i.e. changing from the original return type $[a]$ to the new return type $[a] \to [a]$. The necessary conversion functions between the two types, the latter of which is sometimes called *Hughes lists* (Hughes, 1986), are defined as follows:

$$\begin{array}{lcl} \textbf{type}\ H\ a & = & [a] \to [a] \\ \mathit{rep} & :: & [a] \to H\ a \\ \mathit{rep}\ xs & = & (xs +\!\!+) \\ \mathit{abs} & :: & H\ a \to [a] \\ \mathit{abs}\ h & = & h\ [\,] \end{array}$$

Note that *rep* is just a synonym for $(+\!\!+)$. It is straightforward to verify the worker/wrapper assumption $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{[a]}$. We also have the important property that *rep* forms a monoid homomorphism from lists to Hughes lists, in the sense that:

$$\begin{array}{lcl} \mathit{rep}\ (xs +\!\!+ ys) & = & \mathit{rep}\ xs \circ \mathit{rep}\ ys \\ \mathit{rep}\ [\,] & = & \mathit{id}_{[a]} \end{array}$$

In the case of reverse, it turns out that the most convenient condition to use as the basis for constructing the worker function is condition (2):

$$\mathit{rep}\ (\mathit{snoc}\ x\ xs) \quad = \quad g\ x\ (\mathit{rep}\ xs)$$

We calculate a function g satisfying this equation as follows:

$$\begin{array}{ll} & \mathit{rep}\ (\mathit{snoc}\ x\ xs) \\ = & \quad \{\ \text{applying } \mathit{snoc}\ \} \\ & \mathit{rep}\ (xs +\!\!+ [x]) \\ = & \quad \{\ \mathit{rep} \text{ is a homomorphism}\ \} \\ & \mathit{rep}\ xs \circ \mathit{rep}\ [x] \\ = & \quad \{\ \text{applying } \mathit{rep}\ \} \\ & \mathit{rep}\ xs \circ (x:) \\ = & \quad \{\ \text{define } g\ x\ h = h \circ (x:)\ \} \\ & g\ x\ (\mathit{rep}\ xs) \end{array}$$

Now that we have satisfied the necessary preconditions, applying the worker/wrapper transformation for *fold* gives the following new definitions:

$$\begin{array}{lcl} \mathit{rev} & :: & [a] \to [a] \\ \mathit{rev} & = & \mathit{abs} \circ \mathit{work} \\ \mathit{work} & :: & [a] \to H\ a \\ \mathit{work} & = & \mathit{fold}\ g\ (\mathit{rep}\ [\,]) \end{array}$$

Finally, if we make the list arguments explicit, and then expand out the component functions, we obtain the expected linear time version of reverse that uses an accumulator:

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

We conclude with a number of observations about the above derivation. First of all, in common with the previous derivation of fast reverse using the worker/wrapper technique for *fix* (Gill & Hutton, 2009), once we have made the decision to use Hughes' representation of lists, the rest of the derivation proceeds using simple equational reasoning, without the need for induction. However, in contrast to the previous derivation, using the additional structure afforded by using *fold* avoids the need for the additional functions *wrap* and *unwrap*, the use of worker/wrapper fusion, and the need to expand out the worker as an essential step in the derivation, resulting in a simpler derivation.

4.2 Example: fast reverse revisited

It is interesting now to return to our earlier question of why we don't always use condition (1), which provides an explicit definition for g as a starting point. In the case of the reverse example, the initial definition would then be as follows:

$$g\ x\ y \quad = \quad \mathit{rep}\ (\mathit{snoc}\ x\ (\mathit{abs}\ y))$$

The problem comes when we try and simplify this definition:

$$\begin{array}{ll} & g\ x\ y \\ = & \quad \{\ \text{applying } g\ \} \\ & \mathit{rep}\ (\mathit{snoc}\ x\ (\mathit{abs}\ y)) \\ = & \quad \{\ \text{applying } \mathit{snoc}\ \} \\ & \mathit{rep}\ (\mathit{abs}\ y +\!\!+ [x]) \\ = & \quad \{\ \mathit{rep} \text{ is a homomorphism}\ \} \\ & \mathit{rep}\ (\mathit{abs}\ y) \circ \mathit{rep}\ [x] \\ = & \quad \{\ \text{applying } \mathit{rep}\ \} \\ & \mathit{rep}\ (\mathit{abs}\ y) \circ (x{:}) \end{array}$$

Now we appear to be stuck. We'd like to fuse together *rep* and *abs* in the final expression to give the definition $g\ x\ y = y \circ (x{:})$, but unfortunately it is not the case that $\mathit{rep} \circ \mathit{abs} = \mathit{id}_{H\,a}$.

In order to make progress, we begin by rewriting the worker

$$work \;=\; fold\ g\ (rep\ [\,])$$

by making the first list argument explicit, expanding out the *fold*, and using the above simplification of g to give the following definition using explicit recursion:

$$\begin{array}{lcl} work\ [\,] & = & rep\ [\,] \\ work\ (x : xs) & = & rep\ (abs\ (work\ xs)) \circ (x:) \end{array}$$

While $rep \circ abs = id_{Ha}$ is not true in general, for the special case of values produced by worker itself we do have $rep \circ abs \circ work = work$, the worker/wrapper fusion property (6) from section 3.1, which allows use to rewrite the worker as

$$\begin{array}{lcl} work\ [\,] & = & rep\ [\,] \\ work\ (x : xs) & = & work\ xs \circ (x:) \end{array}$$

which can then be expanded to give the expected definition:

$$\begin{array}{lcl} work\ [\,]\ ys & = & ys \\ work\ (x : xs)\ ys & = & work\ xs\ (y : ys) \end{array}$$

However, an unsatisfactory aspect of the above derivation is the need to rewrite the worker using explicit recursion in order to make progress by applying worker/wrapper fusion. Can the derivation also be performed at the *fold* level, without expanding out the recursion? The key to achieving this is to observe that in this context the second argument of g will always be of the form $rep\ z$ for some list z, since both the base and recursive case for the worker have an application of *rep* at the outer level. Using this assumption, the definition for g can then be simplified as follows:

$$\begin{array}{ll} & g\ x\ y \\ = & \quad \{\ \text{previous simplification}\ \} \\ & rep\ (abs\ y) \circ (x:) \\ = & \quad \{\ \text{assuming}\ y = rep\ z\ \} \\ & rep\ (abs\ (rep\ z)) \circ (x:) \\ = & \quad \{\ abs \circ rep = id\ \} \\ & rep\ z \circ (x:) \\ = & \quad \{\ \text{assuming}\ y = rep\ z\ \} \\ & y \circ (x:) \end{array}$$

Avoiding the need for this kind of ad-hoc additional reasoning is precisely the benefit that we obtain by starting from condition (2) rather than (1). In particular, using $rep\ (f\ x\ y) = g\ x\ (rep\ y)$ as our specification for g makes *explicit* from the outset that we can assume the second argument to g is always of the form $rep\ y$.

5 Worker/wrapper for expressions

For our next example we move from the type of lists to a simple language of expressions comprising integers and addition, together with its associated *fold* operator:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Expr} & = & \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ \mathit{fold} & :: & (a \to a \to a) \to (\mathit{Int} \to a) \to \mathit{Expr} \to a \\ \mathit{fold}\ f\ v\ (\mathit{Val}\ n) & = & v\ n \\ \mathit{fold}\ f\ v\ (\mathit{Add}\ x\ y) & = & f\ (\mathit{fold}\ f\ v\ x)\ (\mathit{fold}\ f\ v\ y) \end{array}$$

Suppose now that we wish to change the return type of a function $\mathit{fold}\ f\ v :: \mathit{Expr} \to a$ from the original type a to some other type b, and that we are given conversion functions $\mathit{rep} :: a \to b$ and $\mathit{abs} :: b \to a$ such that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_a$. In this context, our general theory from section 3 states that any of the three conditions

$$(1) \quad g\ x\ y = \mathit{rep}\ (f\ (\mathit{abs}\ x)\ (\mathit{abs}\ y))$$

$$(2) \quad \mathit{rep}\ (f\ x\ y) = g\ (\mathit{rep}\ x)\ (\mathit{rep}\ y)$$

$$(3) \quad f\ (\mathit{abs}\ x)\ (\mathit{abs}\ y) = \mathit{abs}\ (g\ x\ y)$$

is sufficient to justify the following factorisation of the original *fold* that produces a result of type a into the composition of a worker *fold* that produces a result of type b, and a wrapper function that converts the result back to the original type a:

$$\mathit{fold}\ f\ v \quad = \quad \mathit{abs} \circ \mathit{fold}\ g\ (\mathit{rep} \circ v)$$

5.1 Example: continuation-passing evaluation

Consider the problem of transforming an evaluator for expressions into continuation-passing style, the typical first step in deriving an efficient abstract machine (Hutton & Wright, 2006). Using explicit recursion, an evaluation function can be defined by

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Expr} \to \mathit{Int} \\ \mathit{eval}\ (\mathit{Val}\ n) & = & n \\ \mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathit{eval}\ x + \mathit{eval}\ y \end{array}$$

or equivalently, using the *fold* operator for expressions:

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Expr} \to \mathit{Int} \\ \mathit{eval} & = & \mathit{fold}\ (+)\ \mathit{id} \end{array}$$

Rewriting this definition in continuation-passing style involves taking a function on integers (the continuation) as an extra argument, which using currying corresponds to changing from the original return type Int to the new return type $(\mathit{Int} \to \mathit{Int}) \to \mathit{Int}$. The necessary

conversion functions between the two types are defined as follows:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Cint} & = & (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\ \mathit{rep} & :: & \mathit{Int} \to \mathit{Cint} \\ \mathit{rep}\ n & = & \lambda c \to c\ n \\ \mathit{abs} & :: & \mathit{Cint} \to \mathit{Int} \\ \mathit{abs}\ f & = & f\ \mathit{id} \end{array}$$

It is easy to show that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{\mathit{Int}}$. As with fast reverse, the appropriate starting point for constructing the worker in this case is condition (2),

$$\mathit{rep}\ (x+y) \quad = \quad g\ (\mathit{rep}\ x)\ (\mathit{rep}\ y)$$

from which we calculate a function g satisfying this equation as follows:

$$\begin{array}{ll} & \mathit{rep}\ (x+y)\ c \\ = & \quad \{\ \text{applying } \mathit{rep}\ \} \\ & c\ (x+y) \\ = & \quad \{\ \text{abstracting over } x\ \} \\ & (\lambda n \to c\ (n+y))\ x \\ = & \quad \{\ \text{unapplying } \mathit{rep}\ \} \\ & \mathit{rep}\ x\ (\lambda n \to c\ (n+y)) \\ = & \quad \{\ \text{abstracting over } y\ \} \\ & \mathit{rep}\ x\ (\lambda n \to (\lambda m \to c\ (n+m))\ y) \\ = & \quad \{\ \text{unapplying } \mathit{rep}\ \} \\ & \mathit{rep}\ x\ (\lambda n \to \mathit{rep}\ y\ (\lambda m \to c\ (n+m))) \\ = & \quad \{\ \text{define } g\ a\ b = a\ (\lambda n \to b\ (\lambda m \to c\ (n+m)))\ \} \\ & g\ (\mathit{rep}\ x)\ (\mathit{rep}\ y) \end{array}$$

Now that we have satisfied the necessary preconditions, applying the worker/wrapper transformation for *fold* gives the following definitions

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Expr} \to \mathit{Int} \\ \mathit{eval} & = & \mathit{abs} \circ \mathit{work} \\ \mathit{work} & :: & \mathit{Expr} \to \mathit{Cint} \\ \mathit{work} & = & \mathit{fold}\ g\ (\mathit{rep} \circ \mathit{id}) \end{array}$$

which expand out to give the expected continuation-passing evaluator:

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Expr} \to \mathit{Int} \\ \mathit{eval}\ e & = & \mathit{work}\ e\ \mathit{id} \\ \mathit{work} & :: & \mathit{Expr} \to (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\ \mathit{work}\ (\mathit{Val}\ n)\ c & = & c\ n \\ \mathit{work}\ (\mathit{Add}\ x\ y)\ c & = & \mathit{work}\ x\ (\lambda n \to \mathit{work}\ y\ (\lambda m \to c\ (n+m))) \end{array}$$

Once again, note that the derivation proceeds using simple equational reasoning and does not require induction. Moreover, in contrast to our previous derivation of such an evaluator using more elementary techniques (Hutton & Wright, 2006), using worker/wrapper condition (2) as the starting point results in derivation whose goal is made explicit from the

outset, namely to construct a function g such that $rep\ (x+y) = g\ (rep\ x)\ (rep\ y)$, rather than this property being implicit in the structure of the derivation itself.

6 Efficient substitution

For our final example, we consider a more challenging problem: improving the performance of monadic substitution on trees. The example is take from (Voigtländer, 2008), but whereas the author only sketches a proof of correctness and conjectures that a formal proof may require sophisticated techniques, we show that a simple proof is possible using our worker/wrapper technique for *fold*. We begin by defining the type *Tree a* of binary trees whose leaves contain values of some parameter type *a*:

$$\textbf{data}\ Tree\ a \quad = \quad Leaf\ a \mid Node\ (Tree\ a)\ (Tree\ a)$$

Now recall that in Haskell, the categorical notion of a *monad* is captured by the following class declaration, which states that a parameterised type *m* is a member of the class *Monad* of monadic types if it is equipped with *return* and $\gg\!=$ functions of the specified types:

$$\begin{array}{l}\textbf{class}\ Monad\ m\ \textbf{where}\\ \quad \begin{array}{lcl} return & :: & a \to m\ a \\ (\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \end{array}\end{array}$$

The two functions must also satisfy identity and associativity properties:

$$\begin{array}{lcl} return\ x \gg\!= f & = & f\ x \\ e \gg\!= return & = & e \\ (e \gg\!= f) \gg\!= g & = & e \gg\!= (\lambda x \to f\ x \gg\!= g) \end{array}$$

It is straightforward to make *Tree* into a monadic type by the following instance declaration, and to verify that the required monad laws are satisfied:

$$\begin{array}{l}\textbf{instance}\ Monad\ Tree\ \textbf{where}\\ \quad \begin{array}{lcl} return & :: & a \to Tree\ a \\ return\ x & = & Leaf\ x \\ (\gg\!=) & :: & Tree\ a \to (a \to Tree\ b) \to Tree\ b \\ (Leaf\ x) \gg\!= f & = & f\ x \\ (Node\ l\ r) \gg\!= f & = & Node\ (l \gg\!= f)\ (r \gg\!= f) \end{array}\end{array}$$

This declaration implements the well-known idea that substitution is monadic. In particular, if we view values of type *Tree a* as terms with variables of type *a*, then *return* converts a value into the corresponding term, and $t \gg\!= f$ is the term that results from applying the substitution f to every variable in the term t. For example, given the tree of characters

$$t \quad = \quad Node\ (Leaf\ \texttt{'a'})\ (Leaf\ \texttt{'b'})$$

and the substitution

$$\begin{array}{lcl} f & :: & Char \to Tree\ Int \\ f\ \texttt{'a'} & = & Leaf\ 1 \\ f\ \texttt{'b'} & = & Node\ (Leaf\ 2)\ (Leaf\ 3) \end{array}$$

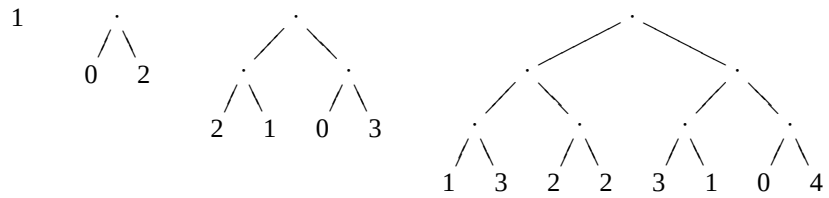
then the expression $t \gg\!= f$ produces the following tree of integers:

$$Node\ (Leaf\ 1)\ (Node\ (Leaf\ 2)\ (Leaf\ 3))$$

Now consider the following recursive function on natural numbers, which uses substitution to produce a tree of integers of a specified depth:

$$\begin{array}{lcl} fullTree & :: & Int \to Tree\ Int \\ fullTree\ 1 & = & return\ 1 \\ fullTree\ (n+1) & = & fullTree\ n \gg\!= \lambda i \to \\ & & Node\ (return\ (n-i))\ (return\ (i+1)) \end{array}$$

That is, a tree of depth 1 is produced by returning a leaf, and a tree of depth $n+1$ by recursively building a tree of depth n, and then using substitution to replace each leaf value i by a tree of depth two with leaf values $n-i$ and $i+1$. For example, the first four trees produced by applying *fullTree* can be pictured as follows:



As we would expect from these examples, *fullTree* takes exponential time. Now consider the function *zigzag* that follows a path down a tree that alternates between moving left (*zig*) and right (*zag*), and returns the resulting leaf value:

$$\begin{array}{lcl} zigzag & :: & Tree\ a \to a \\ zigzag & = & zig \\ & & \textbf{where} \\ & & \quad zig\ (Leaf\ x) = x \\ & & \quad zig\ (Node\ l\ r) = zag\ l \\ & & \quad zag\ (Leaf\ x) = x \\ & & \quad zag\ (Node\ l\ r) = zag\ r \end{array}$$

In a lazy language such as Haskell, evaluating $zigzag\ (fullTree\ n)$ only builds as much of the intermediate tree as necessary to produce the final result, which in this case is a single path. However, due to the iterative nature of *fullTree*, in which the complete tree is potentially traversed at each step in order to increase the depth by one, such an evaluation still requires quadratic time, even in a lazy language. How can this be reduced to linear time?

6.1 The codensity monad

Voigtländer's solution (2008) is based upon changing the representation of trees, using a generalised form of continuation. Recall that a continuation can be viewed as a function that is applied to the result of another computation. Using this idea, we can represent a value x as the function $\lambda c \to c\ x$ that takes a continuation c, and applies this function to x

in order to produce the final result. This representation gives rise to the type $(a \to r) \to r$ of continuation computations of type a that return results of type r:

$$\textbf{type}\ \mathit{Cont}\ r\ a \quad = \quad (a \to r) \to r$$

It is easy to show that *Cont r* is a monadic type. Moreover, we can also parameterise the declaration by another monad *m* to give a *monad transformer* (Liang *et al.*, 1995):

$$\textbf{type}\ \mathit{ContT}\ r\ m\ a \quad = \quad (a \to m\ r) \to m\ r$$

For the purposes of improving the efficiency of *fullTree*, we will use the following generalisation, known as the *codensity* monad transformer (Jaskelioff, 2009):

$$\textbf{type}\ \mathit{CodT}\ m\ a \quad = \quad \forall r.\ ((a \to m\ r) \to m\ r)$$

That is, the result type r is moved from the the left-side of the declaration to the right-side, by exploiting Haskell's notion of *rank 2* types (Peyton Jones *et al.*, 2007). Moving the quantification in this manner means that whereas the continuation monad *ContT r m* has a fixed result type r, the codensity monad *CodT m* has a variable (polymorphic) result type. Making *CodT* into a monad transformer proceeds as follows:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{CodT}\ m)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return} & :: & a \to \mathit{CodT}\ m\ a \\
\mathit{return}\ x & = & \lambda c \to c\ x \\
(\gg\!=) & :: & \mathit{CodT}\ m\ a \to (a \to \mathit{CodT}\ m\ b) \to \mathit{CodT}\ m\ b \\
f \gg\!= g & = & \lambda c \to f\ (\lambda x \to g\ x\ c)
\end{array}
\end{array}$$

Using the codensity monad transformer, we now define a new representation for trees, together with the necessary conversion functions between the original and new types:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Coden}\ a & = & \mathit{CodT}\ \mathit{Tree}\ a \\
\mathit{rep} & :: & \mathit{Tree}\ a \to \mathit{Coden}\ a \\
\mathit{rep}\ t & = & (t \gg\!=) \\
\mathit{abs} & :: & \mathit{Coden}\ a \to \mathit{Tree}\ a \\
\mathit{abs}\ c & = & c\ \mathit{return}
\end{array}$$

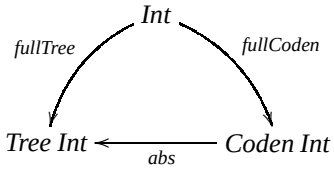
It is interesting to note the similarity to the definitions $\mathit{rep}\ xs = (xs{+}{+})$ and $\mathit{abs}\ f = f\ [\,]$ given earlier for lists. The above definitions for trees have the same structure, except that the monoid operations $+\!+$ and $[\,]$ are generalised to the monad operations $\gg\!=$ and *return*. A simple calculation verifies the worker/wrapper assumption $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{\mathit{Tree}\,a}$:

$$\begin{array}{ll}
 & \mathit{abs}\ (\mathit{rep}\ t) \\
= & \quad \{\ \text{applying } \mathit{rep}\ \} \\
 & \mathit{abs}\ (t \gg\!=) \\
= & \quad \{\ \text{applying } \mathit{abs}\ \} \\
 & t \gg\!= \mathit{return} \\
= & \quad \{\ \text{monad laws}\ \} \\
 & t
\end{array}$$

□

6.2 The term type

To improve the performance of *fullTree*, our aim now is to factorise this function into the composition of a more efficient worker that produces a result in our codensity monad, and a wrapper that converts the result back into the tree monad. That is, we seek to define a function *fullCoden* that makes the following diagram commute:



Following the lead of our previous examples, we might expect to proceed by defining *fullTree* as a *fold* over the type of natural numbers, and then applying our worker/wrapper technique to derive the required worker. For this example, however, it turns out to be preferable to begin by reformulating the problem in terms of a more structured type than the natural numbers. Consider once again the definition for *fullTree*:

$$
\begin{array}{lcl}
\mathit{fullTree} & :: & \mathit{Int} \to \mathit{Tree\ Int} \\
\mathit{fullTree}\ 1 & = & \mathit{return}\ 1 \\
\mathit{fullTree}\ (n+1) & = & \mathit{fullTree}\ n \gg\!= \lambda i \to \\
 & & \mathit{Node}\ (\mathit{return}\ (n-i))\ (\mathit{return}\ (i+1))
\end{array}
$$

In this definition, the resulting trees are built using three functions:

$$
\begin{array}{lcl}
\mathit{return} & :: & a \to \mathit{Tree}\ a \\
(\gg\!=) & :: & \mathit{Tree}\ a \to (a \to \mathit{Tree}\ b) \to \mathit{Tree}\ b \\
\mathit{Node} & :: & \mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Tree}\ a
\end{array}
$$

Based upon this observation, we can define the following type of *tree terms* whose values represent trees that are built using these functions:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Term}\ a\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{Return} & :: & a \to \mathit{Term}\ a \\
\mathit{Bind} & :: & \mathit{Term}\ a \to (a \to \mathit{Term}\ b) \to \mathit{Term}\ b \\
\mathit{Branch} & :: & \mathit{Term}\ a \to \mathit{Term}\ a \to \mathit{Term}\ a
\end{array}
\end{array}
$$

Reifying functions as data in this manner is sometimes called a deep embedding. Note that because *Bind* involves terms of two different types, *Term a* is a GADT (Peyton Jones *et al.*, 2006). Categorically, defining a *fold* for such types requires moving to a functor category, in which objects are functors and arrows are natural transformations (Johann & Ghani,

2008). In Haskell, the *fold* for terms can be defined as follows:

$$\begin{array}{lcl}
\mathit{fold} & :: & (\forall a.\ a \to f\ a) \to \\
 & & (\forall a\, b.\ f\ a \to (a \to f\ b) \to f\ b) \to \\
 & & (\forall a.\ f\ a \to f\ a \to f\ a) \to \\
 & & (\forall a.\ \mathit{Term}\ a \to f\ a) \\
\mathit{fold}\ r\ b\ n\ (\mathit{Return}\ x) & = & r\ x \\
\mathit{fold}\ r\ b\ n\ (\mathit{Bind}\ t\ g) & = & b\ (\mathit{fold}\ r\ b\ n\ t)\ (\mathit{fold}\ r\ b\ n \circ g) \\
\mathit{fold}\ r\ b\ n\ (\mathit{Branch}\ t\ u) & = & n\ (\mathit{fold}\ r\ b\ n\ t)\ (\mathit{fold}\ r\ b\ n\ u)
\end{array}$$

The use of quantifiers in the type for *fold* reflects the use of natural transformations, which in Haskell correspond to polymorphic functions. To ensure the expected universal property we also require that *Term* and f are functors, but we omit the details here.

Using the *fold* operator for terms, the fact that terms represent trees can now be formalised by defining an evaluation function that simply replaces the syntactic constructors on terms by the corresponding semantic operations on trees:

$$\begin{array}{lcl}
\mathit{eval} & :: & \mathit{Term}\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int} \\
\mathit{eval} & = & \mathit{fold}\ \mathit{return}\ (\gg\!=)\ \mathit{Node}
\end{array}$$

In turn, we can define a version of *fullTree* that produces a term rather than a tree, by replacing the use of the tree operations by the appropriate term constructors:

$$\begin{array}{lcl}
\mathit{fullTerm} & :: & \mathit{Int} \to \mathit{Term}\ \mathit{Int} \\
\mathit{fullTerm}\ 1 & = & \mathit{Return}\ 1 \\
\mathit{fullTerm}\ (n+1) & = & \mathit{fullTerm}\ n\ \text{`}\mathit{Bind}\text{`}\ \lambda i \to \\
 & & \mathit{Branch}\ (\mathit{Return}\ (n-i))\ (\mathit{Return}\ (i+1))
\end{array}$$

A simple inductive proof shows that $\mathit{fullTree} = \mathit{eval} \circ \mathit{fullTerm}$.

Base case:

$$\begin{array}{ll}
 & \mathit{eval}\ (\mathit{fullTerm}\ 1) \\
= & \quad \{\ \text{applying } \mathit{fullTerm}\ \} \\
 & \mathit{eval}\ (\mathit{Return}\ 1) \\
= & \quad \{\ \text{applying } \mathit{eval}\ \} \\
 & \mathit{return}\ 1 \\
= & \quad \{\ \text{applying } \mathit{return}\ \} \\
 & \mathit{Leaf}\ 1 \\
= & \quad \{\ \text{unapplying } \mathit{fullTree}\ \} \\
 & \mathit{fullTree}\ 1
\end{array}$$

Inductive case:

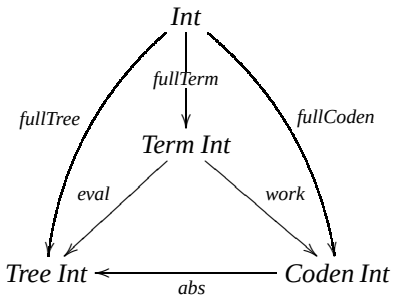
$$\begin{array}{ll}
 & \mathit{eval}\ (\mathit{fullTerm}\ (n+1)) \\
= & \quad \{\ \text{applying } \mathit{fullTerm}\ \} \\
 & \mathit{eval}\ (\mathit{fullTerm}\ n\ \text{`}\mathit{Bind}\text{`}\ \lambda i \to \mathit{Branch}\ (\mathit{Return}\ (n-i))\ (\mathit{Return}\ (i+1))) \\
= & \quad \{\ \text{applying } \mathit{eval}\ \} \\
 & \mathit{eval}\ (\mathit{fullTerm}\ n) \gg\!= \lambda i \to \mathit{Node}\ (\mathit{return}\ (n-i))\ (\mathit{return}\ (i+1))
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ induction hypothesis } \} \\
& \mathit{fullTree}\ n \gg\!= \lambda i \to \mathit{Node}\ (\mathit{return}\ (n-i))\ (\mathit{return}\ (i+1)) \\
= & \{ \text{ applying } \mathit{return} \} \\
& \mathit{fullTree}\ n \gg\!= \lambda i \to \mathit{Node}\ (\mathit{Leaf}\ (n-i))\ (\mathit{Leaf}\ (i+1)) \\
= & \{ \text{ unapplying } \mathit{fullTree} \} \\
& \mathit{fullTree}\ (n+1)
\end{array}
$$

□

6.3 Applying worker/wrapper

Having reformulated *fullTree* using an intermediate type of tree terms, we now seek to complete the following expanded version of our commuting diagram from the previous section, by defining appropriate functions *work* and *fullCoden*:



Commutativity of the left triangle was established in the previous section. The following definition ensures that the right triangle also commutes, by construction:

$$
\begin{array}{lcl}
\mathit{fullCoden} & :: & \mathit{Int} \to \mathit{Coden\ Int} \\
\mathit{fullCoden} & = & \mathit{work} \circ \mathit{fullTerm}
\end{array}
$$

In turn, we define the function *work* using *fold* for terms, by simply supplying the *return* and $\gg\!=$ operations for our codensity monad, and a suitable *node* operation:

$$
\begin{array}{lcl}
\mathit{work} & :: & \mathit{Term\ Int} \to \mathit{Coden\ Int} \\
\mathit{work} & = & \mathit{fold}\ \mathit{return}\ (\gg\!=)\ \mathit{node} \\
\mathit{node} & :: & \mathit{Coden}\ a \to \mathit{Coden}\ a \to \mathit{Coden}\ a \\
\mathit{node}\ f\ g & = & \lambda c \to \mathit{Node}\ (f\ c)\ (g\ c)
\end{array}
$$

To verify that this definition makes the lower triangle in the diagram commute, i.e. $\mathit{eval} = \mathit{abs} \circ \mathit{work}$, we begin by expanding out the definitions for *eval* and *work* to give:

$$\mathit{fold}\ \mathit{return}\ (\gg\!=)\ \mathit{Node} \quad = \quad \mathit{abs} \circ \mathit{fold}\ \mathit{return}\ (\gg\!=)\ \mathit{node}$$

Note that *return* and $\gg\!=$ on the left-side of the equation are for the tree monad, and on the right-side are for the codensity monad. We then apply the worker/wrapper technique for *fold*. In particular, condition (2) for this example expands to give three equations that are

together sufficient to justify the above factorisation:

$$(2.1) \quad rep\ (return\ x) = return\ x$$

$$(2.2) \quad rep\ (t \gg\!= f) = rep\ t \gg\!= rep \circ f$$

$$(2.3) \quad rep\ (Node\ l\ r) = node\ (rep\ l)\ (rep\ r)$$

The first two equations state that *rep* preserves the *return* and $\gg\!=$ operations and is hence a monad morphism, while the last states that *rep* preserves the node operation. Verifying these equations is simply a matter of expanding definitions and using monad laws. We include all three proofs below to emphasise their simplicity.

Proof: (2.1)

$$\begin{array}{ll} & rep\ (return\ x)\ g \\ = & \quad \{\ \text{applying } rep\ \} \\ & return\ x \gg\!= g \\ = & \quad \{\ \text{monad law}\ \} \\ & g\ x \\ = & \quad \{\ \text{unapplying } return \text{ for } Coden\ \} \\ & return\ x\ g \end{array}$$

□

Proof: (2.2)

$$\begin{array}{ll} & rep\ (t \gg\!= f)\ g \\ = & \quad \{\ \text{applying } rep\ \} \\ & (t \gg\!= f) \gg\!= g \\ = & \quad \{\ \text{monad law}\ \} \\ & t \gg\!= (\lambda x \rightarrow f\ x \gg\!= g) \\ = & \quad \{\ \text{unapplying } rep\ \} \\ & t \gg\!= (\lambda x \rightarrow rep\ (f\ x)\ g) \\ = & \quad \{\ \text{unapplying } rep\ \} \\ & rep\ t\ (\lambda x \rightarrow rep\ (f\ x)\ g) \\ = & \quad \{\ \text{unapplying} \gg\!= \text{ for } Coden\ \} \\ & (rep\ t \gg\!= rep \circ f)\ g \end{array}$$

□

Proof: (2.3)

$$\begin{array}{ll} & rep\ (Node\ l\ r)\ g \\ = & \quad \{\ \text{applying } rep\ \} \\ & Node\ l\ r \gg\!= g \\ = & \quad \{\ \text{applying} \gg\!= \text{ for } Tree\ \} \\ & Node\ (l \gg\!= g)\ (r \gg\!= g) \\ = & \quad \{\ \text{unapplying } rep\ \} \\ & Node\ (rep\ l\ g)\ (rep\ r\ g) \\ = & \quad \{\ \text{unapplying } node\ \} \end{array}$$

$$node\ (rep\ l)\ (rep\ r)\ g$$

□

Finally, because the three internal triangles in the diagram commute, the external triangle also commutes, which verifies the desired worker/wrapper factorisation:

$$\begin{array}{lcl} fullTree & :: & Int \to Tree\ Int \\ fullTree & = & abs \circ fullCoden \end{array}$$

Proof:

$$\begin{array}{ll} & fullTree \\ = & \quad \{ \text{ left triangle } \} \\ & eval \circ fullTerm \\ = & \quad \{ \text{ lower triangle } \} \\ & abs \circ work \circ fullTerm \\ = & \quad \{ \text{ right triangle } \} \\ & abs \circ fullCoden \end{array}$$

□

Returning to our original problem of improving the efficiency of $zigzag\ (fullTree\ n)$, if we now replace the original definition for *fullTree* by the new version obtained using the worker/wrapper technique, the time complexity is reduced from quadratic to linear. For example, in a simple experiment using the Glasgow Haskell Compiler the time for $n = 10000$ was reduced from around 90 seconds to 0.2 seconds. If desired, the definition $fullCoden = work \circ fullTerm$ can also be fused to eliminate the use of the intermediate term structure, further reducing the running time to under 0.1 seconds.

We conclude with a few remarks about this example. First of all, despite using a sophisticated optimisation technique in the form of the codensity monad, the proof of correctness of the efficient version of *fullTerm* still only requires simple equational reasoning. Secondly, our proof of the worker/wrapper factorisation $eval = work \circ abs$ is not specific to tree terms built using *fullTerm*, but shows how to optimise the evaluation of *any* such terms. And finally, the use of tree terms also provides an explanation for *why* the optimisation is correct, in the sense that it makes explicit the key idea of implementing the *return*, $\gg\!=$, and *Node* operations on expression trees using the codensity monad.

7 Conclusion and further work

In this article we developed a general worker/wrapper theory for changing the type of recursive functions defined using fold operators, and showed how it can be used in practice as an equational reasoning technique for improving the performance of functional programs. The approach requires only basic categorical and equational reasoning principles, and using fold operators results in simpler and more structured calculations than the previous worker/wrapper theory based upon fixed point operators.

It is also interesting to recount how this work was developed. Initially we focused on the special case of *fold* for lists, and identified conditions (1) and (2) for the case of lists. However, it was not clear how these conditions were related, nor how they related to fold fusion (3), or worker/wrapper fusion (6). It was only when we generalised from lists to an

arbitrary type using initial algebra semantics that it became clear that there were in fact four relevant properties, related by a simple lattice structure. Focusing on lists made it difficult to "see the wood for the trees", and the move to a categorical approach revealed the simple underlying algebraic structure of the problem.

There are many interesting topics for further work, including mechanising the technique, other recursion operators such as *unfold*, weaker versions of the worker/wrapper assumption $abs \circ rep = id$, and other application areas. The monadic substitution example also suggests a new approach to program optimisation that we are particularly keen to explore, based upon a deep embedding of the operations to be optimised and the use of the worker/wrapper technique to demonstrate correctness of the optimised program.

References

Bird, Richard, & de Moor, Oege. (1997). *Algebra of Programming*. Prentice Hall.

Gibbons, Jeremy. (2006). Fission for Program Comprehension. *Pages 162–179 of:* Uustalu, Tarmo (ed), *Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 4014. Springer-Verlag.

Gill, Andy, & Hutton, Graham. (2009). The Worker/Wrapper Transformation. *Journal of Functional Programming*, **19**(2), 227–251.

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. *Information Processing Letters*, **22**(3).

Hutton, Graham. (1999). A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, **9**(4), 355–372.

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

Hutton, Graham, & Wright, Joel. (2006). Calculating an Exceptional Machine. Loidl, Hans-Wolfgang (ed), *Trends in Functional Programming volume 5*. Intellect. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

Jaskelioff, Mauro. (2009). Modular Monad Transformers. *Pages 64–79 of: Proceedings of the European Symposium on Programming*. LNCS, vol. 5502. Springer.

Johann, Patricia, & Ghani, Neil. (2008). Foundations for Structured Programming with GADTs. *Pages 297–308 of: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Liang, Sheng, Hudak, Paul, & Jones, Mark. (1995). Monad Transformers and Modular Interpreters. *Pages 333–343 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Malcolm, Grant. (1990). Algebraic Data Types and Program Transformation. *Science of Computer Programming*, **14**(2-3), 255–280.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. Hughes, John (ed), *Proceedings of the Conference on Functional Programming and Computer Architecture*. LNCS, no. 523. Springer-Verlag.

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

Peyton Jones, Simon, & Launchbury, John. (1991). Unboxed Values as First Class Citizens in a Non-strict Functional Language. *Proceedings of the Conference on Functional Programming and Computer Architecture*. Cambridge, Massachussets: Springer-Verlag.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Washburn, Geoffrey. (2006). Simple Unification-Based Type Inference for GADTs. *Pages 50–61 of: Proceedings of the 11th acm sigplan international conference on functional programming*. ACM Press.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Shields, Mark. (2007). Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming*, **17**(1), 1–82.

Voigtländer, Janis. (2008). Asymptotic Improvement of Computations over Free Monads. *Pages 388–403 of: Proceedings of the 9th international conference on mathematics of program construction*. LNCS, vol. 5133. Marseille, France: Springer-Verlag.