JFP **20** (3 & 4): 353–373, 2010. © Cambridge University Press 2010
doi:10.1017/S0956796810000122 First published online 30 June 2010

Factorising folds for faster functions

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

MAURO JASKELIOFF
Universidad Nacional de Rosario, Rosario, Argentina
(*e-mail:* mauro@fceia.unr.edu.ar)

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

Abstract

The worker/wrapper transformation is a general technique for improving the performance of recursive programs by changing their types. The previous formalisation (A. Gill & G. Hutton, *J. Funct. Program.*, vol. 19, 2009, pp. 227–251) was based upon a simple fixed-point semantics of recursion. In this paper, 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 used for many years, particularly in optimising 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 transformation has been formalised, proved correct, and presented as a general technique for improving the performance of programs by improving the choice of data structures (Gill & Hutton 2009).

The previous formalisation was based upon a simple fixed-point semantics of recursive programs. In this paper, we take a more structured approach, based upon initial-algebra semantics. In particular, we develop a general worker/wrapper theory for changing the type of recursive programs defined using fold operators, and show how it can be used in practice as an equational reasoning technique for improving the performance of programs. More precisely, this paper 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 use of initial algebras also means that our worker/wrapper technique for folds is generic in the underlying recursive type to which it applies (Backhouse *et al.* 1999). That is, the technique is defined and proved once for an arbitrary recursive type, and can then simply be instantiated as required for each new type.

This paper is aimed at readers who are familiar with the basics of initial-algebra semantics (in particular, the concepts of categories, functors, products, co-products, and initial algebras), say to the level of chapter two of Bird & de Moor (1997), but no previous experience with the worker/wrapper transformation is assumed. An extended version of this paper that includes all the proofs is available from the authors' Web pages.

2 Initial-algebra semantics

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

Suppose 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 $f : FA \to A$. In turn, 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:

$$\begin{tikzcd} FA \arrow[r, "Fh"] \arrow[d, "f"'] & FB \arrow[d, "g"] \\ A \arrow[r, "h"'] & B \end{tikzcd}$$

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:

$$\begin{tikzcd} F\mu F \arrow[r, "F\,(fold\ f)"] \arrow[d, "in"'] & FA \arrow[d, "f"] \\ \mu F \arrow[r, dotted, "fold\ f"'] & A \end{tikzcd}$$

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

$$h = fold\ f \quad \Leftrightarrow \quad h \circ in = f \circ Fh$$

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:

$$h \circ f = g \circ Fh \quad \Rightarrow \quad h \circ fold\ f = fold\ g$$

As a simple example of initial-algebra semantics, suppose we define a functor F on the category **SET** by $FA = 1 + A$. Then F has an initial algebra, given by the set $\mathbb{N}$ of natural numbers, together with a function $[zero, succ] : 1 + \mathbb{N} \to \mathbb{N}$ comprising two constructors $zero : 1 \to \mathbb{N}$ and $succ : \mathbb{N} \to \mathbb{N}$ for this set. In turn, given any other set A and functions $v : 1 \to A$ and $f : A \to A$, the function $fold\ [v, f] : \mathbb{N} \to A$ is uniquely defined by

$$\begin{aligned} h\ (zero\ ()) &= v\ () \\ h\ (succ\ n) &= f\ (h\ n) \end{aligned}$$

That is, $fold\ [v, f]$ processes a natural number by replacing the *zero* constructor by the function v, and each *succ* constructor by the function f. For example, a doubling function can be defined by $double = fold\ [zero, succ \circ succ]$, and fusion can then be used to show that $double \circ double = fold\ [zero, succ \circ succ \circ succ \circ succ]$.

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

$$\begin{tikzcd} A \arrow[r, bend left, "rep"] & B \arrow[l, bend left, "abs"] \end{tikzcd}$$

with the property that

$$abs \circ rep = id_A$$

This equation states that *abs* is a left inverse (or retraction) of *rep*, or in more practical terms, that converting a value of the original type into the new type and then back again does not change the value. In the terminology of data representation (Hoare 1972), this means that the *abstract* type A can be faithfully represented by the *concrete* type B. For example, in the case of the category **SET**, the equation ensures that the set A is isomorphic to the subset of B given by the image of *rep*. Given the above assumptions, we now seek conditions under which the following diagram commutes:

$$\begin{tikzcd} & \mu F \arrow[dl, "fold\ f"'] \arrow[dr, "fold\ g"] & \\ A & & B \arrow[ll, "abs"] \end{tikzcd}$$

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* in the manner above, namely that $f \circ F\, abs = abs \circ g$. However, given the assumption that $abs \circ rep = id_A$, we can, in fact, identify four relevant conditions, given by the four possible ways of completing the following commuting diagram that relates the argument algebras f and g:

$$\begin{tikzcd} FA \arrow[r, dash, "F?"] \arrow[d, "f"'] & FB \arrow[d, "g"] \\ A \arrow[r, dash, "?"'] & B \end{tikzcd}$$

by replacing each? in the diagram with either $rep : A \to B$ or $abs : B \leftarrow A$:

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

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

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

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

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:

$$\begin{tikzcd}
 & (4) & \\
(2) \arrow[ur, Rightarrow] & & (3) \arrow[ul, Rightarrow] \\
 & (1) \arrow[ul, Rightarrow] \arrow[ur, Rightarrow] &
\end{tikzcd}$$

It is now straightforward to verify that each of the first three conditions implies the desired factorisation result, namely that $fold\ f = abs \circ fold\ g$. The situation regarding (4) is more involved, 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 manner similar 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 purposes of examples we will primarily focus on (2). Nonetheless, it is interesting to consider the other conditions, and their relationships.

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 $\mathit{fold}\ f = \mathit{abs} \circ \mathit{fold}\ g$, 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:

$$\mathit{fold}\ f = \mathit{abs} \circ \mathit{fold}\ (g \circ F\,(\mathit{rep} \circ \mathit{abs}))$$

The additional term $F\,(\mathit{rep} \circ \mathit{abs})$ in the worker plays the role of a *normalisation* function that is applied after each recursive call. In general, $\mathit{rep} \circ \mathit{abs} \neq \mathit{id}_B$, but we can think of $\mathit{rep} \circ \mathit{abs}$ 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 does (4) imply that $\mathit{fold}\ f = \mathit{abs} \circ \mathit{fold}\ g$. The answer is given by the following condition, which states that $\mathit{rep} \circ \mathit{abs}$ is a homomorphism from g to itself:

$$(5) \quad \mathit{rep} \circ \mathit{abs} \circ g = g \circ F\,(\mathit{rep} \circ \mathit{abs})$$

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 implies the worker/wrapper factorisation.

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*:

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

That is, even though $\mathit{rep} \circ \mathit{abs} = \mathit{id}_B$ 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.

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[1] (Peyton Jones 2003). Our first example concerns lists, for which the *fold* operator in Haskell is defined as follows:

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

That is, the function $\mathit{fold}\ f\ v$ 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 $\mathit{sum} = \mathit{fold}\ (+)\ 0$. The Haskell definition above is equivalent to the categorical definition of *fold* for lists,

[1] Technically, we view Haskell as a meta-language for the category **SET**, which admits simple equational reasoning without the need to consider $\perp$. However, using the 'fast and loose' approach of Danielsson *et al.* (2006), our reasoning is also valid for the total fragment of **CPO**.

except that it uses two arguments f and v rather than combining these as a single argument.

Now suppose we are given a function $\mathit{fold}\ f\ v :: [a] \to b$ 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 $\mathit{rep} :: b \to c$ and $\mathit{abs} :: c \to b$ satisfying the equation $\mathit{abs} \circ \mathit{rep} = \mathit{id}_b$. Then instantiating our general theory from the previous section, we find that any of the three conditions

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

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

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

is sufficient to justify the following factorisation of the original *fold* that produces a result of type 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:

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

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

```
rev          :: [a] → [a]
rev []       = []
rev (x : xs) = rev xs ++ [x]
```

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

```
rev       :: [a] → [a]
rev       = fold snoc []

snoc      :: a → [a] → [a]
snoc x xs = xs ++ [x]
```

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 set-up. 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:

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

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

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

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):

$$rep\ (snoc\ x\ xs) = g\ x\ (rep\ xs)$$

We calculate a function g satisfying this equation as follows:

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

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

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

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

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

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, the additional structure made explicit 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 now interesting to return to our earlier question of why we do not 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:

$$g\ x\ y = rep\ (snoc\ x\ (abs\ y))$$

The problem comes when we try and simplify this definition:

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

Now we appear to be stuck. We would 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 $rep \circ abs = id_{H\,a}$. 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:

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

While $rep \circ abs = id_{H\,a}$ 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), which allows us to rewrite the worker as

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

which can then be expanded to give the expected definition:

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

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 cases 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:

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

Now suppose we wish to change the return type of a function $fold\ f\ v :: Expr \to a$ from the original type a to some other type b, and that we are given conversion functions $rep :: a \to b$ and $abs :: b \to a$ such that $abs \circ rep = id_a$. In this context, our general worker/wrapper theory states that any of the three conditions

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

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

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

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:

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

Now 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

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

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

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

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:

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

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):

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

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

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

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

$$\begin{aligned} &eval && :: Expr \rightarrow Int \\ &eval && = abs \circ work \\[4pt] &work && :: Expr \rightarrow Cint \\ &work && = fold\ g\ (rep \circ id) \end{aligned}$$

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

$$\begin{aligned} &eval && :: Expr \rightarrow Int \\ &eval\ e && = work\ e\ id \\[4pt] &work && :: Expr \rightarrow (Int \rightarrow Int) \rightarrow Int \\ &work\ (Val\ n)\ c && = c\ n \\ &work\ (Add\ x\ y)\ c && = work\ x\ (\lambda n \rightarrow work\ y\ (\lambda m \rightarrow c\ (n + m))) \end{aligned}$$

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 a 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 taken 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 generalising the type *Expr* from the previous section to the type *Tree a* of binary trees with leaves of type *a*:

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

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:

$$\begin{aligned} &\mathbf{class}\ Monad\ m\ \mathbf{where} \\ &\quad return \quad :: \quad a \rightarrow m\ a \\ &\quad (\gg\!=) \quad :: \quad m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \end{aligned}$$

The two functions must also satisfy identity and associativity properties:

$$\begin{aligned} return\ x \gg\!= f &= f\ x \\ e \gg\!= return &= e \\ (e \gg\!= f) \gg\!= g &= e \gg\!= (\lambda x \rightarrow f\ x \gg\!= g) \end{aligned}$$

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:

```
instance Monad Tree where
   return           :: a → Tree a
   return x         = Leaf x

   (≫=)             :: Tree a → (a → Tree b) → Tree b
   (Leaf x) ≫= f    = f x
   (Node l r) ≫= f  = Node (l ≫= f) (r ≫= f)
```

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 = Node (Leaf 'a') (Leaf 'b')
```

and the substitution

```
f      :: Char → Tree Int
f 'a'  = Leaf 1
f 'b'  = Node (Leaf 2) (Leaf 3)
```

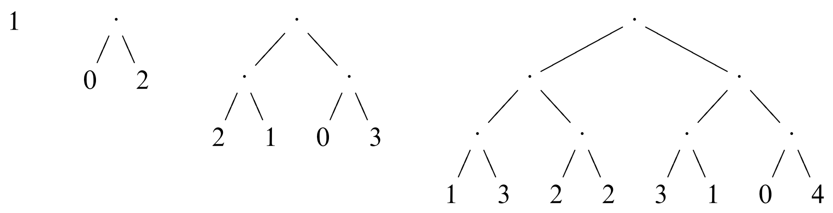
the expression $t \gg= f$ produces the following tree of integers:

```
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:

```
fullTree          :: Int → Tree Int
fullTree 1        = return 1
fullTree (n + 1)  = fullTree n ≫= λi →
                       Node (return (n − i)) (return (i + 1))
```

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 2 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:

```
zigzag :: Tree a → a
zigzag = zig
         where
           zig (Leaf x) = x
           zig (Node l r) = zag l
           zag (Leaf x) = x
           zag (Node l r) = zig r
```

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 1, 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 (2008) solution is based upon changing the representation of trees, using the notion of continuations. 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 $\lambda c \rightarrow c\ x$ 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 $(a \rightarrow r) \rightarrow r$ of continuation computations of type a that return results of type r:

```
type Cont r a = (a → r) → 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):

```
type ContT r m a = (a → m r) → m r
```

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

```
type CodT m a = ∀r. ((a → m r) → m r)
```

That is, the result type r is moved from the left-hand side of the declaration to the right-hand 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:

```
instance Monad m ⇒ Monad (CodT m) where
  return    :: a → CodT m a
  return x  = λc → c x

  (≫=)      :: CodT m a → (a → CodT m b) → CodT m b
  f ≫= g    = λc → f (λx → g x c)
```

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:

```
type Coden a = CodT Tree a

rep       :: Tree a → Coden a
rep t     = (t ≫=)

abs       :: Coden a → Tree a
abs c     = c return
```

It is interesting to note the similarity to the definitions $rep\ xs = (xs +\!\!+)$ and $abs\ f = f\ [\,]$ 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 $abs \circ rep = id_{Tree\ a}$.

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 the 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:

$$\begin{tikzcd}
 & Int \arrow[dl, bend right, "fullTree"'] \arrow[dr, bend left, "fullCoden"] & \\
Tree\ Int & & Coden\ Int \arrow[ll, "abs"]
\end{tikzcd}$$

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*:

```
fullTree           :: Int → Tree Int
fullTree 1         = return 1
fullTree (n + 1)   = fullTree n ≫= λi →
                        Node (return (n − i)) (return (i + 1))
```

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

$$\begin{aligned}
&\mathit{return} &&:: a \rightarrow \mathit{Tree}\ a\\
&(\gg\!=) &&:: \mathit{Tree}\ a \rightarrow (a \rightarrow \mathit{Tree}\ b) \rightarrow \mathit{Tree}\ b\\
&\mathit{Node} &&:: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{data}\ \mathit{Term}\ a\ \mathbf{where}\\
&\quad \mathit{Return} :: a \rightarrow \mathit{Term}\ a\\
&\quad \mathit{Bind}\ \ \ :: \mathit{Term}\ a \rightarrow (a \rightarrow \mathit{Term}\ b) \rightarrow \mathit{Term}\ b\\
&\quad \mathit{Branch} :: \mathit{Term}\ a \rightarrow \mathit{Term}\ a \rightarrow \mathit{Term}\ a
\end{aligned}$$

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:

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

The use of quantifiers in the type for *fold* reflects the use of natural transformations, which correspond to polymorphic functions in Haskell. 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:

$$\begin{aligned}
&\mathit{eval} :: \mathit{Term}\ a \rightarrow \mathit{Tree}\ a\\
&\mathit{eval} = \mathit{fold}\ \mathit{return}\ (\gg\!=)\ \mathit{Node}
\end{aligned}$$

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:

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

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

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*:

$$\begin{tikzcd}
 & \mathit{Int} \arrow[d, "\mathit{fullTerm}"] \arrow[ddl, bend right, "\mathit{fullTree}"'] \arrow[ddr, bend left, "\mathit{fullCoden}"] & \\
 & \mathit{Term\ Int} \arrow[dl, "\mathit{eval}"'] \arrow[dr, "\mathit{work}"] & \\
\mathit{Tree\ Int} & & \mathit{Coden\ Int} \arrow[ll, "\mathit{abs}"]
\end{tikzcd}$$

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

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

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:

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

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

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

Note that *return* and $\gg=$ on the left-hand side of the equation are for the tree monad, and on the right-hand 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:

$$(2.1)\quad \mathit{rep}\ (\mathit{return}\ x) = \mathit{return}\ x$$

$$(2.2)\quad \mathit{rep}\ (t \gg= f) = \mathit{rep}\ t \gg= \mathit{rep} \circ f$$

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

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)

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

□

Proof : (2.2)

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

□

Proof : (2.3)

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

□

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

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

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 $n = 10,000$ was reduced from around 90 to 0.2 s. 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 s.

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 paper, 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.

Acknowledgements

We would like to thank Ralf Hinze and the four anonymous referees for their useful comments. The last author would like to thank the Functional Programming Lab in Nottingham for funding a one-week research visit during June 2009.

References

Backhouse, R., Jansson, P., Jeuring, J. & Meertens, L. (1999) Generic programming: An introduction. In *Advanced Functional Programming*, Swierstra, D., Henriques, P. & Oliveira, J. (eds), LNCS 1608. Springer-Verlag, Berlin, pp. 28–115.

Bird, R. & de Moor, O. (1997) *Algebra of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Danielsson, N. A., Gibbons, J., Hughes, J. & Jansson, P. (2006) Fast and loose reasoning is morally correct. In *Principles of Programming Languages*. ACM Press, New York.

Gibbons, J. (2006) Fission for program comprehension. In *Mathematics of Program Construction*, Uustalu, T. (ed), Lecture Notes in Computer Science, vol. 4014. Springer-Verlag, Berlin, pp. 162–179.

Gill, A. & Hutton, G. (2009) The worker/wrapper transformation, *J. Funct. Program.*, 19 (2): 227–251.

Hoare, T. (1972) Proof of correctness of data representations, *Acta Inform.*, 1 (4): 271–281.

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

Hutton, G. (1999) A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9 (4): 355–372.

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

Hutton, G. & Wright, J. (2006) Calculating an exceptional machine. In *Trends in Functional Programming*, vol. 5, Loidl, H.-W. (ed), Intellect, UK. Selected papers from the *Fifth Symposium on Trends in Functional Programming*, Munich, Germany, November 2004.

Jaskelioff, M. (2009) Modular monad transformers. In *Proceedings of the European Symposium on Programming*. LNCS, vol. 5502. Springer-Verlag, Berlin, pp. 64–79.

Johann, P. & Ghani, N. (2008) Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, pp. 297–308.

Liang, S., Hudak, P. & Jones, M. (1995) Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, pp. 333–343.

Malcolm, G. (1990) Algebraic data types and program transformation. *Sci. Comput. Program.*, 14 (2–3): 255–280.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, Hughes, J. (ed), LNCS, vol. 523. Springer-Verlag, Berlin.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK. Available at: `www.haskell.org/definition`

Peyton Jones, S. & Launchbury, J. (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, Berlin.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, pp. 50–61.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Shields, M. (2007) Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17 (1): 1–82.

Voigtländer, J. (2008) Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction.* LNCS, vol. 5133. Marseille, France: Springer-Verlag, Berlin, pp. 388–403.