The Under-Performing Unfold

A new approach to optimising corecursive programs

Jennifer Hackett Graham Hutton
University of Nottingham
{jph,gmh}@cs.nott.ac.uk

Mauro Jaskelioff
Universidad Nacional de Rosario, Argentina
CIFASIS–CONICET, Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

This paper presents a new approach to optimising corecursive programs by factorisation. In particular, we focus on programs written using the corecursion operator unfold. We use and expand upon the proof techniques of guarded coinduction and unfold fusion, capturing a pattern of generalising coinductive hypotheses by means of abstraction and representation functions. The pattern we observe is simple, has not been observed before, and is widely applicable. We develop a general program factorisation theorem from this pattern, demonstrating its utility with a range of practical examples.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Theory, Proof Methods, Optimisation

Keywords fusion, factorisation, coinduction, unfolds

1. Introduction

When writing programs that produce data structures it is often natural to use the technique of *corecursion* [8, 20], in which subterms of the result are produced by recursive calls. This is particularly useful in lazy languages such as Haskell, as it allows us to process parts of the result without producing the rest of it. In this way, we can write programs that deal with large or infinite data structures and trust that the memory requirements remain reasonable.

However, while this technique may allow us to save on the number of recursive calls by producing data lazily, the *time cost* of each call or the *space cost* of the arguments can still be a problem. For this reason, it is necessary to examine various approaches to reducing these costs. A commonly-used approach is *program fusion* [4, 11, 31], where separate stages of a computation are combined into one to avoid producing intermediate data. For this paper, our focus will be on the opposite technique of *program factorisation* [7], where a computation is split into separate parts. In particular, we consider the problem of splitting a computation into the combination of a more efficient *worker* program that uses a different representation of data and a *wrapper* program that effects the necessary change of representation [10].

The primary contribution of this paper is a general factorisation theorem for programs in the form of an *unfold* [9], a common pattern of corecursive programming. By exploiting the categorical principle of duality, we adapt the theory of *worker-wrapper factorisation for folds*, which was developed initially by Hutton, Jaskelioff and Gill [15] and subsequently extended by Sculthorpe and Hutton [24]. We discuss the practical considerations of the dual theory, which differ from those of the original theory and avoid the need for strictness side conditions. In addition, we revise the theory to further improve its utility. As we shall see, the resulting theory for unfolds captures a common pattern of coinductive reasoning that is simple, has not been observed before, and is widely applicable. We demonstrate the application of the new theory with a range of practical examples of varying complexity.

The development of our new theory is first motivated using a small programming example, which we then generalise using category theory. We use only simple concepts for this generalisation, and only a minimal amount of categorical knowledge is required. Readers without a background in category theory should not be deterred. The primary application area for our theory is functional languages such as Haskell, however the use of categorical concepts means that our theory is more widely applicable.

2. Coinductive Types and Proofs

Haskell programmers will be familiar with recursive type definitions, where a type is defined in terms of itself. For example, the type of natural numbers can be defined as follows:

$$\mathbf{data}\ \mathbb{N} = \mathit{Zero} \mid \mathit{Succ}\ \mathbb{N}$$

This definition states that an element of the type $\mathbb{N}$ is either Zero or $\mathit{Succ}\ n$ for some n that is also an element of the type $\mathbb{N}$. More formally, recursive type definitions can be given meaning as fixed points of type-level equations. For example, we can read the above as defining $\mathbb{N}$ to be a fixed point of the equation $X = 1 + X$, where 1 is the unit type and $+$ is the disjoint sum of types.

Assuming that such a fixed point equation has at least one solution, there are two we will typically be interested in. First of all, there is the *least* fixed point, which is given by the smallest type that is closed under the constructors. This is known as an *inductive* type. In a set-theoretic context, the inductive interpretation of our definition of $\mathbb{N}$ is simply the set of natural numbers, with constructors Zero and Succ having the usual behaviour.

Alternatively, there is the *greatest* fixed point, which is given by the largest type that supports deconstruction of values by pattern matching. This is known as a *coinductive* type [12]. In a set-theoretic context, the coinductive interpretation of our definition of $\mathbb{N}$ is the set of naturals augmented with an infinite value ∞ that is the solution to the equation $\infty = \mathit{Succ}\ \infty$.

In general, coinductively-defined sets have infinite elements, while inductively-defined sets do not. In the setting of Haskell, however, types correspond to *(pointed) complete partial orders*

(CPOs) rather than sets, where there is no distinction between inductive and coinductive type definitions as the two notions coincide [6]. For the purposes of this article we will use Haskell syntax as a metalanguage for programming in both set-theoretic and CPO contexts. It will therefore be necessary to distinguish between inductive and coinductive definitions, which we do so by using the keywords **data** and **codata** respectively.

For example, we could define an inductive type of lists and a coinductive type of streams as follows, using the constructor (:) in both definitions for consistency with Haskell usage:

$$\begin{aligned}&\mathbf{data}\ \ [a] &&= a : [a] \mid [\,] \\ &\mathbf{codata}\ \mathit{Stream}\ a &&= a : \mathit{Stream}\ a\end{aligned}$$

If types are sets, the first definition gives finite lists while the second gives infinite streams. If types are CPOs, the first definition gives both finite and infinite lists, while the second gives infinite streams. Also note that in the context of sets, if streams were defined using **data** rather than **codata** the resulting type would be empty.

2.1 Coinduction

To reason about elements of inductive types one can use the technique of induction. Likewise, to reason about elements of coinductive types one can use the dual technique of *coinduction*. To formalise this precisely involves the notion of *bisimulation* [8, 12]. In this section we shall give an informal presentation of *guarded* coinduction [3, 28], a special case that avoids the need for such machinery. This form of coinduction is closely related to the unique fixed point principle developed by Hinze [14].

To prove an equality $\mathit{lhs} = \mathit{rhs}$ between expressions of the same coinductive type using guarded coinduction, we simply attempt the proof in the usual way using equational reasoning. However, we may also make use of the *coinductive hypothesis*, which allows us to substitute lhs for rhs (or vice-versa) provided that we only do so immediately underneath a constructor of the coinductive type. We say that such a use of the coinductive hypothesis is *guarded*. For example, if we define the following functions that produce values of type $\mathit{Stream}\ \mathbb{N}$

$$\begin{aligned}&\mathit{from}\ n &&= n : \mathit{from}\ (n+1) \\ &\mathit{skips}\ n &&= n : \mathit{skips}\ (n+2) \\ &\mathit{double}\ (n : \mathit{ns}) &&= n * 2 : \mathit{double}\ \mathit{ns}\end{aligned}$$

then we can show that $\mathit{skips}\ (n*2) = \mathit{double}\ (\mathit{from}\ n)$ for any natural number n using guarded coinduction:

$$\begin{aligned}&\mathit{skips}\ (n*2) \\ =\ &\{\ \text{definition of } \mathit{skips}\ \} \\ &n*2 : \mathit{skips}\ ((n*2)+2) \\ =\ &\{\ \text{arithmetic}\ \} \\ &n*2 : \mathit{skips}\ ((n+1)*2) \\ =\ &\{\ \text{coinductive hypothesis}\ \} \\ &n*2 : \mathit{double}\ (\mathit{from}\ (n+1)) \\ =\ &\{\ \text{definition of } \mathit{double}\ \} \\ &\mathit{double}\ (n : \mathit{from}\ (n+1)) \\ =\ &\{\ \text{definition of } \mathit{from}\ \} \\ &\mathit{double}\ (\mathit{from}\ n)\end{aligned}$$

Despite the apparent circularity in using an instance of our desired result in the third step of the proof, the proof is guarded because the use of the coinductive hypothesis only occurs directly below the (:) constructor. Therefore the reasoning is valid.

3. Example: Tabulating a Function

Consider the problem of *tabulating* a function $f :: \mathbb{N} \to a$ by applying it to every natural number in turn and forming a stream from the results. We would like to define a function that performs this task, specified informally as follows:

$$\begin{aligned}&\mathit{tabulate} :: (\mathbb{N} \to a) \to \mathit{Stream}\ a \\ &\mathit{tabulate}\ f = [f\ 0, f\ 1, f\ 2, f\ 3, \ldots]\end{aligned}$$

The following definition satisfies this specification:

$$\mathit{tabulate}\ f = f\ 0 : \mathit{tabulate}\ (f \circ (+1))$$

However, this definition is inefficient, as with each recursive call the function argument becomes more costly to apply, as shown in the following expansion of the definition:

$$\mathit{tabulate}\ f = [f\ 0, (f \circ (+1))\ 0, (f \circ (+1) \circ (+1))\ 0, \ldots]$$

The problem is that the natural number is recomputed from scratch each time by repeated application of $(+1)$ to 0. If we were to save the result and re-use it in future steps, we could avoid repeating work. The idea can be implemented by defining a new function that takes the current value as an additional argument:

$$\begin{aligned}&\mathit{tabulate}' :: (\mathbb{N} \to a, \mathbb{N}) \to \mathit{Stream}\ a \\ &\mathit{tabulate}'\ (f, n) = f\ n : \mathit{tabulate}'\ (f, n+1)\end{aligned}$$

The correctness of the more efficient implementation for tabulation can be captured by the following equation

$$\mathit{tabulate}\ f = \mathit{tabulate}'\ (f, 0)$$

which can written in point-free form as

$$\mathit{tabulate} = \mathit{tabulate}' \circ (\lambda f \to (f, 0))$$

This equation can be viewed as a *program factorisation*, in which $\mathit{tabulate}$ is factored into the composition of $\mathit{tabulate}'$ and the function $\lambda f \to (f, 0)$. This latter function effects a *change of data representation* from the old argument type $\mathbb{N} \to a$ to the new argument type $(\mathbb{N} \to a, \mathbb{N})$. In order to try to prove the above equation, we proceed by guarded coinduction:

$$\begin{aligned}&\mathit{tabulate}\ f \\ =\ &\{\ \text{definition of } \mathit{tabulate}\ \} \\ &f\ 0 : \mathit{tabulate}\ (f \circ (+1)) \\ =\ &\{\ \text{coinduction hypothesis}\ \} \\ &f\ 0 : \mathit{tabulate}'\ (f \circ (+1), 0) \\ =\ &\{\ \text{assumption}\ \} \\ &f\ 0 : \mathit{tabulate}'\ (f, 1) \\ =\ &\{\ \text{definition of } \mathit{tabulate}'\ \} \\ &\mathit{tabulate}'\ (f, 0)\end{aligned}$$

To complete the proof, the assumption used in the third step $\mathit{tabulate}'\ (f \circ (+1), 0) = \mathit{tabulate}'\ (f, 1)$ must be verified. We could attempt to prove this as follows:

$$\begin{aligned}&\mathit{tabulate}'\ (f \circ (+1), 0) \\ =\ &\{\ \text{definition of } \mathit{tabulate}'\ \} \\ &(f \circ (+1))\ 0 : \mathit{tabulate}'\ (f \circ (+1), 1) \\ =\ &\{\ \text{composition, arithmetic}\ \} \\ &f\ 1 : \mathit{tabulate}'\ (f \circ (+1), 1) \\ =\ &\{\ \text{assumption}\ \} \\ &f\ 1 : \mathit{tabulate}'\ (f, 2) \\ =\ &\{\ \text{definition of } \mathit{tabulate}'\ \} \\ &\mathit{tabulate}'\ (f, 1)\end{aligned}$$

Once again, however, the proof relies on an assumption that needs to be verified. We could continue like this *ad infinitum* without ever actually completing the proof! We can avoid this problem by *generalising* our correctness property to

$$\mathit{tabulate}\ (f \circ (+n)) = \mathit{tabulate}'\ (f, n)$$

which in the case of $n = 0$ simplifies to the original equation. The proof of the generalised property is now a straightforward application of guarded coinduction, with no assumptions required:

$$\begin{array}{ll}
 & \mathit{tabulate}\ (f \circ (+n)) \\
= & \{ \text{definition of } \mathit{tabulate} \} \\
 & (f \circ (+n))\ 0\ :\ \mathit{tabulate}\ (f \circ (+n) \circ (+1)) \\
= & \{ \text{simplification} \} \\
 & f\ n\ :\ \mathit{tabulate}\ (f \circ (+(n+1))) \\
= & \{ \text{coinduction hypothesis} \} \\
 & f\ n\ :\ \mathit{tabulate}'\ (f, n+1) \\
= & \{ \text{definition of } \mathit{tabulate}' \} \\
 & \mathit{tabulate}'\ (f, n)
\end{array}$$

It is often necessary to generalise coinductive hypotheses in this way for proofs by guarded coinduction, just as it is often necessary to generalise inductive hypotheses for proofs by induction.

4. Abstracting

The above example is an instance of a general pattern of optimisation that is simple yet powerful. We abstract from this example to the general case in two steps, firstly by generalising on the underlying datatypes involved and secondly by generalising on the pattern of corecursive definition that is used.

4.1 Abstracting on Datatypes

In the tabulation example, we replaced the original function of type $(\mathbb{N} \rightarrow a) \rightarrow \mathit{Stream}\ a$ with a more efficient function of type $(\mathbb{N} \rightarrow a, \mathbb{N}) \rightarrow \mathit{Stream}\ a$, changing the type of the argument. Essentially, we used a "larger" type as a representation of a "smaller" type. We can generalise this idea to any two types where one serves as a representation of the other.

Suppose we have two types, a and b, with conversion functions $\mathit{abs} :: b \rightarrow a$ and $\mathit{rep} :: a \rightarrow b$ such that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_a$. We can think of b as a larger type that faithfully represents the elements of the smaller type a. Now suppose that we are given a function $\mathit{old} :: a \rightarrow c$, together with a more efficient version $\mathit{new} :: b \rightarrow c$ that acts on the larger type b. Then the correctness of the more efficient version can be captured by the equation

$$\mathit{old} = \mathit{new} \circ \mathit{rep}$$

However, using the assumption that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_a$ we can strengthen this property by the following calculation:

$$\begin{array}{ll}
 & \mathit{old} = \mathit{new} \circ \mathit{rep} \\
\Leftrightarrow & \{ \mathit{abs} \circ \mathit{rep} = \mathit{id}_a \} \\
 & \mathit{old} \circ \mathit{abs} \circ \mathit{rep} = \mathit{new} \circ \mathit{rep} \\
\Leftarrow & \{ \text{cancelling } \mathit{rep} \text{ on both sides} \} \\
 & \mathit{old} \circ \mathit{abs} = \mathit{new}
\end{array}$$

In summary, if we wish to show that function new is correct, it suffices to show that $\mathit{old} \circ \mathit{abs} = \mathit{new}$. This stronger correctness property may be easier to prove than the original version.

We now apply the above idea to our earlier example. In this case, the appropriate abs and rep functions are:

$$\begin{array}{l}
\mathit{abs} :: (\mathbb{N} \rightarrow a, \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow a) \\
\mathit{abs}\ (f, n) = f \circ (+n) \\[1ex]
\mathit{rep} :: (\mathbb{N} \rightarrow a) \rightarrow (\mathbb{N} \rightarrow a, \mathbb{N}) \\
\mathit{rep}\ f = (f, 0)
\end{array}$$

The required relationship $\mathit{abs} \circ \mathit{rep} = \mathit{id}$ follows immediately from the fact that 0 is the identity for addition. The above calculation can therefore be specialised to our example as follows:

$$\begin{array}{ll}
 & \forall f\ .\ \mathit{tabulate}\ f = \mathit{tabulate}'\ (f, 0) \\
\Leftrightarrow & \{ \text{definition of } \mathit{rep} \} \\
 & \forall f\ .\ \mathit{tabulate}\ f = \mathit{tabulate}'\ (\mathit{rep}\ f) \\
\Leftrightarrow & \{ \text{composition, extensionality} \} \\
 & \mathit{tabulate} = \mathit{tabulate}' \circ \mathit{rep} \\
\Leftrightarrow & \{ \mathit{abs} \circ \mathit{rep} = \mathit{id} \} \\
 & \mathit{tabulate} \circ \mathit{abs} \circ \mathit{rep} = \mathit{tabulate}' \circ \mathit{rep} \\
\Leftarrow & \{ \text{cancelling } \mathit{rep} \text{ on both sides} \} \\
 & \mathit{tabulate} \circ \mathit{abs} = \mathit{tabulate}' \\
\Leftrightarrow & \{ \text{composition, extensionality} \} \\
 & \forall f, n\ .\ \mathit{tabulate}\ (\mathit{abs}\ (f, n)) = \mathit{tabulate}'\ (f, n) \\
\Leftrightarrow & \{ \text{definition of } \mathit{abs} \} \\
 & \forall f, n\ .\ \mathit{tabulate}\ (f \circ (+n)) = \mathit{tabulate}'\ (f, n)
\end{array}$$

The final equation is precisely the generalised correctness property from the previous section, but has now been obtained from an abstract framework that is generic in the underlying datatypes.

4.2 Abstracting on Corecursion Pattern

For the next step in the generalisation, we make some assumptions about the corecursive structure of the functions that we are dealing with. In particular, we assume that they are instances of a specific pattern of corecursive definition called *unfold* [9, 18].

4.2.1 Unfold for Streams

An unfold is a function that produces an element of a coinductive type as its result, producing all subterms of the result using recursive calls. This pattern can be abstracted into an operator, which we define in the case of streams as follows:

$$\begin{array}{l}
\mathrm{unfold} :: (a \rightarrow b) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow \mathit{Stream}\ b \\
\mathrm{unfold}\ h\ t\ x = h\ x\ :\ \mathrm{unfold}\ h\ t\ (t\ x)
\end{array}$$

The function unfold $h\ t$ produces a stream from a seed value x by using the function h to produce the head of the stream from the seed, and applying the function t to produce a new seed that is used to generate the tail of the stream in the same manner. For efficiency we could choose to combine the h and t functions into a single function of type $a \rightarrow (b, a)$, allowing for increase sharing between the two computations. However, we present a 'tuple-free' version because it leads to simpler equational reasoning.

By providing suitable definitions for h and t, it is straightforward to redefine the functions $\mathit{tabulate}$ and $\mathit{tabulate}'$:

$$\begin{array}{lll}
\mathit{tabulate} & = \mathrm{unfold}\ h\ t & \\
 & \quad \mathbf{where}\ h\ f & = f\ 0 \\
 & \quad \phantom{\mathbf{where}}\ t\ f & = f \circ (+1) \\
\mathit{tabulate}' & = \mathrm{unfold}\ h'\ t' & \\
 & \quad \mathbf{where}\ h'\ (f, n) & = f\ n \\
 & \quad \phantom{\mathbf{where}}\ t'\ (f, n) & = (f, n+1)
\end{array}$$

In this way, unfold allows us to factor out the basic steps in the computations. A similar unfold operator can be defined for any coinductive type. For example, for infinite binary trees

$$\mathbf{codata}\ \mathit{Tree}\ a = \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

the following definition for unfold $l\ n\ r$ produces a tree from a seed value by using l and r to produce new seeds for the left and right subtrees, and n to produce the node value:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{unfold} :: (a \rightarrow a) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow \mathit{Tree}\ b} \\
\mathrm{unfold}\ l\ n\ r\ x = \mathit{Node} & (\mathrm{unfold}\ l\ n\ r\ (l\ x)) \\
 & (n\ x) \\
 & (\mathrm{unfold}\ l\ n\ r\ (r\ x))
\end{array}$$

Once again, the l, n and r functions could be combined.

4.2.2 Unfold Fusion

The unfold operator for any type has an associated *fusion* law [18], which provides sufficient conditions for when the composition of an unfold with another function can be expressed as a single unfold. In the case of streams, the law is as follows:

Theorem 1 (Unfold Fusion for Streams)**.** *Given*

$$\begin{array}{lll} h :: a \to c & h' :: b \to c & g :: b \to a \\ t :: a \to a & t' :: b \to b & \end{array}$$

we have the following implication:

$$\begin{array}{l} \quad \text{unfold } h\ t \circ g = \text{unfold } h'\ t' \\ \Leftarrow \\ \quad h' = h \circ g \ \wedge \ g \circ t' = t \circ g \end{array}$$

The proof is a simple application of guarded coinduction:

$$\begin{array}{ll} & \text{unfold } h'\ t'\ x \\ = & \{ \text{definition of unfold} \} \\ & h'\ x \ : \ \text{unfold } h'\ t'\ (t'\ x) \\ = & \{ \text{coinduction hypothesis} \} \\ & h'\ x \ : \ \text{unfold } h\ t\ (g\ (t'\ x)) \\ = & \{ \text{first assumption: } h' = h \circ g \} \\ & h\ (g\ x) \ : \ \text{unfold } h\ t\ (g\ (t'\ x)) \\ = & \{ \text{second assumption: } g \circ t' = t \circ g \} \\ & h\ (g\ x) \ : \ \text{unfold } h\ t\ (t\ (g\ x)) \\ = & \{ \text{definition of unfold} \} \\ & \text{unfold } h\ t\ (g\ x) \end{array}$$

The fusion law provides sufficient conditions for when our strengthened correctness property $\mathit{old} \circ \mathit{abs} = \mathit{new}$ holds. Assuming that old and new can both be expressed as unfolds, then:

$$\begin{array}{ll} & \mathit{old} \circ \mathit{abs} = \mathit{new} \\ \Leftrightarrow & \{ \mathit{old} = \text{unfold } h\ t, \mathit{new} = \text{unfold } h'\ t' \} \\ & \text{unfold } h\ t \circ \mathit{abs} = \text{unfold } h'\ t' \\ \Leftarrow & \{ \text{fusion} \} \\ & h' = h \circ \mathit{abs} \ \wedge \ \mathit{abs} \circ t' = t \circ \mathit{abs} \end{array}$$

4.3 Unfold Factorisation for Streams

Combining the two ideas of abstracting on the types and abstracting on the corecursion pattern, we obtain a general theorem for factorising functions defined using unfold for streams.

Theorem 2 (Unfold Factorisation for Streams)**.** *Given*

$$\begin{array}{lll} \mathit{abs} :: b \to a & h :: a \to c & h' :: b \to c \\ \mathit{rep} :: a \to b & t :: a \to a & t' :: b \to b \end{array}$$

satisfying the assumptions

$$\begin{array}{ll} \mathit{abs} \circ \mathit{rep} = \mathit{id}_a & \\ h' & = h \circ \mathit{abs} \\ \mathit{abs} \circ t' & = t \circ \mathit{abs} \end{array}$$

we have the factorisation

$$\text{unfold } h\ t = \text{unfold } h'\ t' \circ \mathit{rep}$$

Using this result, we can split a function unfold h t *into the composition of a* worker *function* unfold h' t' *that uses a different representation of data and a* wrapper *function rep that effects the necessary change of data representation.*

We now apply this to the *tabulate* example. As we have already shown that $\mathit{abs} \circ \mathit{rep} = \mathit{id}$, it is only necessary to verify the remaining two assumptions. We start with the first assumption:

$$\begin{array}{ll} & h\ (\mathit{abs}\ (f, n)) \\ = & \{ \text{definition of } \mathit{abs} \} \\ & h\ (f \circ (+n)) \\ = & \{ \text{definition of } h \} \\ & (f \circ (+n))\ 0 \\ = & \{ \text{simplification} \} \\ & f\ n \\ = & \{ \text{definition of } h' \} \\ & h'\ (f, n) \end{array}$$

Now, the second assumption:

$$\begin{array}{ll} & t\ (\mathit{abs}\ (f, n)) \\ = & \{ \text{definition of } \mathit{abs} \} \\ & t\ (f \circ (+n)) \\ = & \{ \text{definition of } t \} \\ & f \circ (+n) \circ (+1) \\ = & \{ \text{simplification} \} \\ & f \circ (+(n+1)) \\ = & \{ \text{definition of } \mathit{abs} \} \\ & \mathit{abs}\ (f, n+1) \\ = & \{ \text{definition of t'} \} \\ & \mathit{abs}\ (t'\ (f, n)) \end{array}$$

In conclusion, by generalising from our tabulation example we have derived a framework for factorising corecursive functions that are defined using the unfold operator for streams.

5. Categorifying

To recap, we have combined the idea of a change of data representation with an application of fusion to produce a factorisation theorem for stream unfolds. This theorem covers cases not covered by fusion alone. For example, attempting to prove the *tabulate* example correct simply using fusion fails in precisely the same way that our attempted proof using coinduction failed.

However, so far we have only concerned ourselves with the coinductive type of streams. If we wish to apply this technique to other coinductive types, it would seem that we must define unfold and prove its fusion law for every such type we intend to use. Thankfully this is not the case, as category theory provides a convenient generic approach to modelling coinductive types and their unfold operators using the notion of *final coalgebras* [18].

5.1 Final Coalgebras

Suppose that we fix a category $\mathcal{C}$ and a functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ on this category. Then an F-*coalgebra* is a pair (A, f) consisting of an object A along with an arrow $f : A \to \mathsf{F}\ A$. We often omit the object A as it is implicit in the type of f. A *homomorphism* between coalgebras $f : A \to \mathsf{F}\ A$ and $g : B \to \mathsf{F}\ B$ is an arrow $h : A \to B$ such that $\mathsf{F}\ h \circ f = g \circ h$. This property is captured by the following commutative diagram:

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

Intuitively, a coalgebra $f : A \to \mathsf{F}\ A$ can be thought of as giving a *behaviour* to elements of A, where the possible behaviours are specified by the functor F. For example, if we define $\mathsf{F}\ X = 1 + X$ on the category $\mathbf{Set}$ of sets and total functions, then a coalgebra $f : A \to 1 + A$ is the transition function of a state machine in which each element of A is either a terminating state or has a single successor. In turn, a homomorphism corresponds to a behaviour-preserving mapping, in the sense that if we first apply the homomorphism h and then the target behaviour captured by g, we obtain the same result as if we apply the source behaviour captured by f and then apply h to the components of the result.

A *final* coalgebra, denoted $(\nu\mathsf{F}, \mathit{out})$, is an F-coalgebra to which any other coalgebra has a unique homomorphism. If a final coalgebra exists, it is unique up to isomorphism. Given a coalgebra $f : A \rightarrow \mathsf{F}\ A$, the unique homomorphism from f to the final coalgebra out is denoted $\text{unfold}\ f :: A \rightarrow \nu\mathsf{F}$. This *uniqueness property* can be captured by the following equivalence:

$$h = \text{unfold}\, f \ \Leftrightarrow\ \mathsf{F}\ h \circ f = \mathit{out} \circ h$$

We also have a fusion rule for unfold:

Theorem 3 (Unfold Fusion for Final Coalgebras)**.** *Given*

$$f : A \rightarrow \mathsf{F}\,A \quad g : B \rightarrow \mathsf{F}\,B \quad h : A \rightarrow B$$

we have the following implication:

$$\begin{array}{l} \quad \text{unfold}\ g \circ h = \text{unfold}\, f \\ \Leftarrow \\ \quad \mathsf{F}\ h \circ f = g \circ h \end{array}$$

The proof of this theorem can be conveniently captured by the following commutative diagram:

$$\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, "f"'] \arrow[rr, dashed, bend left, "\text{unfold}\ f"] & B \arrow[r, "\text{unfold}\ g"] \arrow[d, "g"] & \nu\mathsf{F} \arrow[d, "\mathit{out}"] \\
\mathsf{F}\ A \arrow[r, "\mathsf{F}\ h"'] & \mathsf{F}\ B \arrow[r, "\mathsf{F}\ (\text{unfold}\ g)"'] & \mathsf{F}\ (\nu\mathsf{F})
\end{tikzcd}$$

The left square commutes by assumption while the right square commutes because $\text{unfold}\ g$ is a coalgebra homomorphism. Therefore, the outer rectangle commutes, meaning that $\text{unfold}\ g \circ h$ is a homomorphism from f to out. Finally, because homomorphisms to the final coalgebra out are unique and $\text{unfold}\ f$ is also such a homomorphism, the result $\text{unfold}\ g \circ h = \text{unfold}\, f$ holds.

We illustrate the above concepts with a concrete example. Consider the functor $\mathsf{F}\ X = \mathbb{N} \times X$ on the category $\mathbf{Set}$. This functor has a final coalgebra $(\mathit{Stream}\ \mathbb{N}, \langle \mathit{head}, \mathit{tail} \rangle)$, consisting of the set $\mathit{Stream}\ \mathbb{N}$ of streams of natural numbers together with the function $\langle \mathit{head}, \mathit{tail} \rangle : \mathit{Stream}\ \mathbb{N} \rightarrow \mathbb{N} \times \mathit{Stream}$ that combines the stream destructors $\mathit{head} : \mathit{Stream}\ \mathbb{N} \rightarrow \mathbb{N}$ and $\mathit{tail} : \mathit{Stream}\ \mathbb{N} \rightarrow \mathit{Stream}\ \mathbb{N}$. Given any set A and functions $h : A \rightarrow \mathbb{N}$ and $t : A \rightarrow A$, the function $\text{unfold}\langle h, t\rangle : A \rightarrow \mathit{Stream}\ \mathbb{N}$ is uniquely defined by the two equations

$$\begin{array}{l} \mathit{head} \circ \text{unfold}\langle h, t\rangle = h \\ \mathit{tail}\ \ \circ \text{unfold}\langle h, t\rangle = \text{unfold}\langle h, t\rangle \circ t \end{array}$$

which are equivalent to the more familiar definition of unfold using using the stream constructor (:) presented earlier:

$$\text{unfold}\ h\ t\ x = h\ x\ :\ \text{unfold}\ h\ t\ (t\ x)$$

We also note that the earlier fusion law for streams is simply a special case of the more general fusion law where $\mathsf{F}\ X = \mathbb{N} \times X$. The fusion precondition simplifies as follows

$$\begin{array}{l} \mathsf{F}\ g \circ \langle h', t' \rangle = \langle h, t\rangle \circ g \\ \Leftrightarrow \quad \{ \text{definition of } \mathsf{F}, \text{products} \} \\ \langle h', g \circ t' \rangle = \langle h \circ g, t \circ g \rangle \\ \Leftrightarrow \quad \{ \text{separating components} \} \\ h' = h \circ g \ \wedge\ g' \circ t = t \circ g \end{array}$$

and the postcondition is clearly equivalent. All of the above also holds for $\mathit{Stream}\ A$ for an arbitrary set A.

It is now straightforward to generalise our earlier unfold factorisation theorem from streams to final coalgebras. Combining the general unfold fusion law with the same type abstraction idea from before, we obtain the following theorem.

Theorem 4 (General Unfold Factorisation)**.** *Given*

$$\begin{array}{ll} \mathit{abs} : B \rightarrow A & f : A \rightarrow \mathsf{F}\,A \\ \mathit{rep} : A \rightarrow B & g : B \rightarrow \mathsf{F}\,B \end{array}$$

satisfying the assumptions

$$\begin{array}{l} \mathit{abs} \circ \mathit{rep} = \mathit{id}_A \\ \mathsf{F}\ \mathit{abs} \circ g = f \circ \mathit{abs} \end{array}$$

we have the factorisation

$$\text{unfold}\, f = \text{unfold}\ g \circ \mathit{rep}$$

that splits the original corecursive program unfold f *into the composition of a worker* unfold g *and a wrapper rep.*

5.2 Exploiting Duality

Our results up to this point have been generic with respect to the choice of a category $\mathcal{C}$. This is helpful, because not only are the results general, they are also subject to *duality*.

In category theory, the principle of duality states that if a property holds of all categories, then the *dual* of that property must also hold of all categories. By applying this duality principle to our general unfold factorisation theorem, we obtain the following factorisation theorem for *folds*, the categorical dual of unfolds:

Theorem 5. *Given*

$$\begin{array}{ll} \mathit{abs} : A \rightarrow B & f : \mathsf{F}\,A \rightarrow A \\ \mathit{rep} : B \rightarrow A & g : \mathsf{F}\,B \rightarrow B \end{array}$$

satisfying the assumptions

$$\begin{array}{l} \mathit{rep} \circ \mathit{abs} = \mathit{id}_A \\ g \circ \mathsf{F}\ \mathit{abs} = \mathit{abs} \circ f \end{array}$$

we have the factorisation

$$\text{fold}\, f = \mathit{rep} \circ \text{fold}\ g$$

that splits the original recursive program fold f *into the composition of a wrapper rep and a worker* fold g*.*

Note that now rep is required to be a left-inverse of abs, rather than the other way around. If we swap their names to reflect this new situation, we see that this is a special case of the following general result, due to Sculthorpe and Hutton [24]:

Theorem 6 (Worker-Wrapper Factorisation for Initial Algebras)**.** *Given*

$$\begin{array}{ll} \mathit{abs} : B \rightarrow A & f : \mathsf{F}\,A \rightarrow A \\ \mathit{rep} : A \rightarrow B & g : \mathsf{F}\,B \rightarrow B \end{array}$$

satisfying one of the assumptions

$$\begin{array}{lll} (A) & \mathit{abs} \circ \mathit{rep} & = \mathit{id}_A \\ (B) & \mathit{abs} \circ \mathit{rep} \circ f & = f \\ (C) & \text{fold}\ (\mathit{abs} \circ \mathit{rep} \circ f) & = \text{fold}\, f \end{array}$$

and one of the conditions

$$\begin{array}{ll} (1)\ g = \mathit{rep} \circ f \circ \mathsf{F}\ \mathit{abs} & (1\beta)\ \text{fold}\ g = \text{fold}\ (\mathit{rep} \circ f \circ \mathsf{F}\ \mathit{abs}) \\ (2)\ g \circ \mathsf{F}\ \mathit{rep} = \mathit{rep} \circ f & (2\beta)\ \text{fold}\ g = \mathit{rep} \circ \text{fold}\, f \\ (3)\ f \circ \mathsf{F}\ \mathit{abs} = \mathit{abs} \circ g & \end{array}$$

we have the factorisation

$$\text{fold } f = abs \circ \text{fold } g$$

If we now apply duality in turn to this theorem, we obtain an even more general version of our unfold factorisation theorem:

Theorem 7 (Worker-Wrapper Factorisation for Final Coalgebras)**.** *Given*

$$\begin{array}{ll} abs : B \to A & f : A \to \mathsf{F}\ A \\ rep : A \to B & g : B \to \mathsf{F}\ B \end{array}$$

satisfying one of the assumptions

$$\begin{array}{ll} (A)\ abs \circ rep & = id_A \\ (B)\ f \circ abs \circ rep & = f \\ (C)\ \text{unfold } (f \circ abs \circ rep) & = \text{unfold } f \end{array}$$

and one of the conditions

$$\begin{array}{l} (1)\ g = \mathsf{F}\ rep \circ f \circ abs \\ (2)\ \mathsf{F}\ abs \circ g = f \circ abs \\ (3)\ \mathsf{F}\ rep \circ f = g \circ rep \end{array}$$

$$\begin{array}{l} (1\beta)\ \text{unfold } g = \text{unfold } (\mathsf{F}\ rep \circ f \circ abs) \\ (2\beta)\ \text{unfold } g = \text{unfold } f \circ abs \end{array}$$

we have the factorisation

$$\text{unfold } f = \text{unfold } g \circ rep$$

At this point it would be reasonable to ask why we did not simply present the dualised theorem straight away. This is indeed possible, but we do not feel it would be a good approach. In particular, our systematic development that starts from a concrete example and then applies steps of abstraction, generalisation and dualisation provides both motivation and explanation for the theorem.

We now turn our attention to interpreting Theorem 7. First of all, assumptions (B) and (C) are simply generalised versions of our original assumption (A), in the sense that (A) ⇒ (B) ⇒ (C). Secondly, conditions (1) and (3) are alternatives to the original condition (2), providing a degree of flexibility for the user of the theorem to select the most convenient. In general these three conditions are unrelated, but any of them is sufficient to ensure that the theorem holds. Finally, the β conditions in the second group arise as weaker versions of the corresponding conditions in the first, i.e. (1) ⇒ (1β) and (2) ⇒ (2β), and given assumption (C) the two β conditions are equivalent. We omit proofs as they are dual to those in [24].

While the new assumptions and conditions are dual to those of the original theorem, they differ significantly in terms of their practical considerations, which we shall discuss now. Firstly, assumption (B) in the theorem for fold, i.e. $abs \circ rep \circ f = f$, can be interpreted as "for any x in the range of f, $abs\ (rep\ x) = x$". This can therefore be proven by reasoning only about such x. When applying the theorem for unfold, this kind of simple reasoning for assumption (B) is not possible, as f is now applied last rather than first and hence cannot be factored out of the proof.

Secondly, condition (2) in the fold case, i.e. $rep \circ f = g \circ \mathsf{F}\ rep$, allowed g to depend on a precondition set up by rep. If such a precondition is desired for the unfold case, condition (3) must be used. This has important implications for use of this theorem as a basis for optimisation, as we will often derive g based on a specification given by one of the conditions.

Finally, we note that proving (C), (1β) or (2β) for the fold case usually requires induction. To prove the corresponding properties for the unfold case will usually require the less widely-understood technique of coinduction. These properties may therefore turn out to be less useful in the unfold case for practical purposes, despite only requiring a technique of comparable complexity. If we want to use this theory to avoid coinduction altogether, assumption (C) and the β conditions are not applicable. Section 5.3 offers a way around this problem in the case of (C).

5.3 Refining Assumption (C)

As it stands, assumption (C) is expressed as an equality between two corecursive programs defined using unfold, and hence may be non-trivial to prove. However, we can derive an equivalent assumption that may be easier to prove in practice:

$$\begin{array}{ll} & \text{unfold } f = \text{unfold } (f \circ abs \circ rep) \\ \Leftrightarrow & \{\ \text{uniqueness property of unfold } (f \circ abs \circ rep)\ \} \\ & out \circ \text{unfold } f = \mathsf{F}\ (\text{unfold } f) \circ f \circ abs \circ rep \\ \Leftrightarrow & \{\ \text{unfold } f \text{ is a homomorphism}\ \} \\ & out \circ \text{unfold } f = out \circ \text{unfold } f \circ abs \circ rep \\ \Leftrightarrow & \{\ out \text{ is an isomorphism}\ \} \\ & \text{unfold } f = \text{unfold } f \circ abs \circ rep \end{array}$$

We denote this equivalent version of assumption (C) as (C'). As this new assumption concerns only the conversions abs and rep along with the original program unfold f, it may be provable simply from the original program's correctness properties.

Assumption (C') also offers a simpler proof of Theorem 7 than one obtains by dualising the proof in [24]. We start from this assumption and use the fact that in this context, conditons (1), (2) and (1β) all imply (2β):

$$\begin{array}{ll} & \text{unfold } f = \text{unfold } f \circ abs \circ rep \\ \Rightarrow & \{\ (2\beta)\text{: unfold } g = \text{unfold } f \circ abs\ \} \\ & \text{unfold } f = \text{unfold } g \circ rep \end{array}$$

The proof in the case of condition (3) remains the same as previously. We conclude by noting that the implication (B) ⇒ (C') is not as obvious as the original implication (B) ⇒ (C). Altering the theory in this manner thus "moves work" from proving the main result to proving the relationships between the conditions.

5.4 Applying the Theory in Haskell

The category that is usually used to model Haskell types and functions is $\mathbf{CPO}$, the category of (pointed) complete partial orders and continuous functions. While a fold operator can be defined in this category, its uniqueness property carries a *strictness* side condition [18]. As a result, the worker-wrapper theory for folds in $\mathbf{CPO}$ requires a strictness condition of its own [24]. However, in the case of unfold in $\mathbf{CPO}$ the uniqueness property holds with no side conditions [18] so our theorem can be freely used to reason about Haskell programs without such concerns.

6. Examples

We now present a collection of worked examples, demonstrating how our new factorisation theorem may be applied. Firstly, we revisit the tabulation example, and show that it is a simple application of the theory. Secondly, we consider the problem of cycling a list, where we reduce the time cost by delaying expensive operations and performing them in a batch. We believe that this will be a common use of our theory. Thirdly, we consider the problem of taking the initial segment of a list, which allows us to demonstrate how sometimes different choices of condition can lead to the same result. Finally, we consider the problem of flattening a tree. In all cases the proofs are largely mechanical and the main inspiration necessary is in the choice of a new data representation.

With the exception of the initial segment example, all of the following examples occur in the context of the category $\mathbf{Set}$ of sets and total functions. Working in $\mathbf{Set}$ results in simpler reasoning as we do not need to consider issues of partiality.

6.1 Example: Tabulating a Function

We can instantiate our theory as shown above to give a proof of the correctness of our tabulate example. The proof uses assumption (A) and condition (2). Therefore we see that this example is a simple application of the worker-wrapper machinery.

6.2 Example: Cycling a List

The function *cycle* takes a non-empty finite list and produces the stream consisting of repetitions of that list. For example:

$$\mathit{cycle}\ [1,2,3] = [1,2,3,1,2,3,1,2,3,\ldots$$

One possible definition for *cycle* is as follows, in which we write $[a]^+$ for the type of non-empty lists of type a:

$$\begin{array}{l}
\mathit{cycle} :: [a]^+ \to \mathit{Stream}\ a \\
\mathit{cycle}\ (x : xs) = x : \mathit{cycle}\ (xs +\!\!+ [x])
\end{array}$$

However, this definition is inefficient, as the append operator $+\!\!+$ takes linear time in the length of the input list. Recalling that $\mathit{Stream}\ \ a$ is the final coalgebra of the functor $\mathsf{F}\ X = a \times X$, we can rewrite *cycle* as an unfold:

$$\begin{array}{l}
\mathit{cycle} = \mathrm{unfold}\ h\ t \\
\quad \mathbf{where}\ h\ xs = \mathit{head}\ xs \\
\qquad\qquad\ \ t\ xs\ = \mathit{tail}\ xs +\!\!+ [\mathit{head}\ xs]
\end{array}$$

The idea we shall apply to improve the performance of *cycle* is to combine several $+\!\!+$ operations into one, thus reducing the average cost. To achieve this, we create a new representation where the original list of type $[a]^+$ is augmented with a (possibly empty) list of elements that have been added to the end. We keep this second list in reverse order so that appending a single element is a constant-time operation. The *rep* and *abs* functions are as follows:

$$\begin{array}{l}
\mathit{rep} :: [a]^+ \to ([a]^+, [a]) \\
\mathit{rep}\ xs = (xs, [\,]) \\
\\
\mathit{abs} :: ([a]^+, [a]) \to [a]^+ \\
\mathit{abs}\ (xs, ys) = xs +\!\!+ \mathit{reverse}\ ys
\end{array}$$

Given these definitions it is easy to verify assumption (A):

$$\begin{array}{ll}
& \mathit{abs}\ (\mathit{rep}\ xs) \\
= & \{\ \text{definition of } \mathit{rep}\ \} \\
& \mathit{abs}\ (xs, [\,]) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
& xs +\!\!+ \mathit{reverse}\ [\,] \\
= & \{\ \text{definition of } \mathit{reverse}\ \} \\
& xs +\!\!+ [\,] \\
= & \{\ [\,] \text{ is unit of } +\!\!+\ \} \\
& xs
\end{array}$$

For this example we take condition (2), i.e. $\mathsf{F}\ \ \mathit{abs} \circ g = f \circ \mathit{abs}$, as our specification of g, once again specialising to the two conditions $h \circ \mathit{abs} = h'$ and $t \circ \mathit{abs} = \mathit{abs} \circ t'$. From this we can calculate h' and t' separately. First we calculate h':

$$\begin{array}{ll}
& h'\ (xs, ys) \\
= & \{\ \text{specification}\ \} \\
& h\ (\mathit{abs}\ (xs, ys)) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
& h\ (xs +\!\!+ \mathit{reverse}\ ys)) \\
= & \{\ \text{definition of } h\ \} \\
& \mathit{head}\ (xs +\!\!+ \mathit{reverse}\ ys) \\
= & \{\ xs \text{ is nonempty}\ \} \\
& \mathit{head}\ xs
\end{array}$$

Now we calculate a definition for t'. Starting from the specification $\mathit{abs} \circ t' = t \circ \mathit{abs}$, we calculate as follows:

$$\begin{array}{ll}
& t\ (\mathit{abs}\ (xs, ys)) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
& t\ (xs +\!\!+ \mathit{reverse}\ ys)) \\
= & \{\ \text{case analysis}\ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to t\ ([x] \qquad\ +\!\!+ \mathit{reverse}\ ys) \\
& \quad (x : xs') \to t\ ((x : xs') +\!\!+ \mathit{reverse}\ ys) \\
= & \{\ \text{definition of } +\!\!+\ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to t\ (x : ([\,] \ \ +\!\!+ \mathit{reverse}\ ys)) \\
& \quad (x : xs') \to t\ (x : (xs' +\!\!+ \mathit{reverse}\ ys)) \\
= & \{\ \text{definition of } t\ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to [\,] \ \ +\!\!+ \mathit{reverse}\ ys +\!\!+ [x] \\
& \quad (x : xs') \to xs' +\!\!+ \mathit{reverse}\ ys +\!\!+ [x] \\
= & \{\ \text{definition of } \mathit{reverse}, +\!\!+\ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to \qquad\quad \mathit{reverse}\ (x : ys) \\
& \quad (x : xs') \to xs' +\!\!+ \mathit{reverse}\ (x : ys) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to \mathit{abs}\ (\mathit{reverse}\ (x : ys), \qquad [\,]) \\
& \quad (x : xs') \to \mathit{abs}\ (xs' \qquad\qquad\quad\ , x : ys) \\
= & \{\ \text{pulling } \mathit{abs} \text{ out of cases}\ \} \\
& \mathit{abs}\ (\mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad\ \to (\mathit{reverse}\ (x : ys) \quad\ , \qquad [\,]) \\
& \quad (x : xs') \to (xs' \qquad\qquad\qquad\ , x : ys))
\end{array}$$

Hence, t' can be defined as follows:

$$\begin{array}{l}
t'\ ([x], ys) \qquad = (\mathit{reverse}\ (x : ys), [\,]) \\
t'\ (x : xs, ys) = (xs, x : ys)
\end{array}$$

In conclusion, by applying our worker-wrapper theorem, we have calculated a factorised version of *cycle*

$$\begin{array}{l}
\mathit{cycle} = \mathrm{unfold}\ h'\ t' \circ \mathit{rep} \\
\quad \mathbf{where}\ h'\ (xs, ys) \qquad = \mathit{head}\ xs \\
\qquad\qquad\quad t'\ ([x], ys) \qquad = (\mathit{reverse}\ (x : ys), [\,]) \\
\qquad\qquad\quad t'\ (x : xs, ys) = (xs, x : ys)
\end{array}$$

which can be written directly as

$$\begin{array}{l}
\mathit{cycle} = \mathit{cycle}' \circ \mathit{rep} \\
\quad \mathbf{where} \\
\qquad \mathit{cycle}'\ ([x], ys) \qquad = x : \mathit{cycle}'\ (\mathit{reverse}\ (x : ys), [\,]) \\
\qquad \mathit{cycle}'\ (x : xs, ys) = x : \mathit{cycle}'\ (xs, x : ys)
\end{array}$$

This version only performs a *reverse* operation once for every cycle of the input list, so the average cost to produce a single element is now constant. We believe that this kind of optimisation — in which costly operations are delayed and combined into a single operation — will be a common use of our theory.

6.3 Example: Initial Segment of a List

The function *init* takes a list and returns the list consisting of all the elements of the original list except the last one:

$$\begin{array}{l}
\mathit{init} :: [a] \to [a] \\
\mathit{init}\ [\,] \qquad\quad = \bot \\
\mathit{init}\ [x] \qquad\ = [\,] \\
\mathit{init}\ (x : xs) = x : \mathit{init}\ xs
\end{array}$$

Here, $\bot$ represents the failure of the function to produce a result. (In Haskell we would not need to give this first case, but we make it explicit here for the purposes of reasoning.)

This example is particularly interesting as it does not require the function being optimised to be explicitly written as an unfold at any point. As the function in question is partial, the relevant category in this case is $\mathbf{CPO}$ rather than $\mathbf{Set}$.

Each call of init checks to see if the argument is empty. However, the argument of the recursive call can never be empty, as if it were then the second case would have been matched rather than the third. We would therefore like to perform this check only once. We can use unfold worker-wrapper to achieve this, by essentially using a "de-consed" list as our representation:

$$\begin{array}{l}
\mathit{rep} :: [a] \to (a, [a]) \\
\mathit{rep}\ [\,] \qquad\quad = \bot \\
\mathit{rep}\ (x : \mathit{xs}) = (x, \mathit{xs}) \\
\\
\mathit{abs} :: (a, [a]) \to [a] \\
\mathit{abs}\ (x, \mathit{xs}) = x : \mathit{xs}
\end{array}$$

In this case, assumption (A) fails:

$$\begin{array}{ll}
 & \mathit{abs}\ (\mathit{rep}\ [\,]) \\
= & \{\ \text{definition of } \mathit{rep}\ \} \\
 & \mathit{abs}\ \bot \\
= & \{\ \mathit{abs} \text{ is strict}\ \} \\
 & \bot \\
\neq & [\,]
\end{array}$$

If we were to rewrite init as an unfold, we could then prove (B). However, we instead avoid this by using the alternative assumption (C'). This expands to $\mathit{init} \circ \mathit{abs} \circ \mathit{rep} = \mathit{init}$, which we prove by case analysis on the argument. For the empty list:

$$\begin{array}{ll}
 & \mathit{init}\ (\mathit{abs}\ (\mathit{rep}\ [\,])) \\
= & \{\ \text{definition of } \mathit{rep}\ \} \\
 & \mathit{init}\ (\mathit{abs}\ \bot) \\
= & \{\ \mathit{abs} \text{ is strict}\ \} \\
 & \mathit{init}\ \bot \\
= & \{\ \mathit{init} \text{ is strict}\ \} \\
 & \bot \\
= & \{\ \text{definition of } \mathit{init}\ \} \\
 & \mathit{init}\ [\,]
\end{array}$$

For the undefined value $\bot$:

$$\begin{array}{ll}
 & \mathit{init}\ (\mathit{abs}\ (\mathit{rep}\ \bot) \\
= & \{\ \mathit{init},\ \mathit{abs} \text{ and } \mathit{rep} \text{ are all strict}\ \} \\
 & \bot
\end{array}$$

Otherwise:

$$\begin{array}{ll}
 & \mathit{init}\ (\mathit{abs}\ (\mathit{rep}\ (x : \mathit{xs}))) \\
= & \{\ \text{definition of } \mathit{rep}\ \} \\
 & \mathit{init}\ (\mathit{abs}\ (x, \mathit{xs})) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
 & \mathit{init}\ (x : \mathit{xs})
\end{array}$$

Using Condition (2β)

Firstly, we demonstrate a derivation of a new worker function that avoids writing init explicitly in terms of unfold. The only condition that permits this is (2β), which expands to $\mathit{init}' = \mathit{init} \circ \mathit{abs}$. We can calculate the definition of init' simply by applying this specification to (x, xs):

$$\begin{array}{ll}
 & \mathit{init}'\ (x, \mathit{xs}) \\
= & \{\ \text{specification of } \mathit{init}'\ \} \\
 & \mathit{init}\ (\mathit{abs}\ (x, \mathit{xs})) \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
 & \mathit{init}\ (x : \mathit{xs}) \\
= & \{\ \text{definition of } \mathit{init}\ \} \\
 & \mathbf{case}\ (x : \mathit{xs})\ \mathbf{of} \\
 & \quad [\,] \qquad\quad \to \bot \\
 & \quad [y] \qquad\ \ \to [\,] \\
 & \quad (y : \mathit{ys}) \to y : \mathit{init}\ \mathit{ys} \\
= & \{\ \text{removing redundant case}\ \} \\
 & \mathbf{case}\ (x : \mathit{xs})\ \mathbf{of} \\
 & \quad [y] \qquad\ \ \to [\,] \\
 & \quad (y : \mathit{ys}) \to y : \mathit{init}\ \mathit{ys} \\
= & \{\ \text{factoring out } x\ \} \\
 & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
 & \quad [\,] \to [\,] \\
 & \quad \mathit{ys} \to x : \mathit{init}\ \mathit{ys} \\
= & \{\ \mathit{ys} \text{ is nonempty}\ \} \\
 & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
 & \quad [\,] \qquad\quad \to [\,] \\
 & \quad (y : \mathit{ys}') \to x : \mathit{init}\ (y : \mathit{ys}') \\
= & \{\ \text{definition of } \mathit{abs}\ \} \\
 & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
 & \quad [\,] \qquad\quad \to [\,] \\
 & \quad (y : \mathit{ys}') \to x : \mathit{init}\ (\mathit{abs}\ (y, \mathit{ys}')) \\
= & \{\ \text{specification of } \mathit{init}'\ \} \\
 & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
 & \quad [\,] \qquad\quad \to [\,] \\
 & \quad (y : \mathit{ys}') \to x : \mathit{init}'\ (y, \mathit{ys}')
\end{array}$$

As we used the specification of init' in its own derivation, the above derivation only guarantees partial correctness. We should take a moment to convince ourselves that the resulting definition does actually satisfy the specification. In this case, both sides are clearly total, so there is no problem. In conclusion, we have derived an alternative definition for the function init

$$\begin{array}{l}
\mathit{init} = \mathit{init}' \circ \mathit{rep} \\
\quad \mathbf{where} \\
\quad\quad \mathit{rep}\ \ [\,] \qquad\qquad = \bot \\
\quad\quad \mathit{rep}\ \ (x : \mathit{xs}) \qquad = (x, \mathit{xs}) \\
\quad\quad \mathit{init}'\ (x, [\,]) \qquad\ \ = [\,] \\
\quad\quad \mathit{init}'\ (x, y : \mathit{ys}) = x : \mathit{init}'\ (y, \mathit{ys})
\end{array}$$

that only performs the check for the empty list once.

Note that we derived the more efficient version of init using worker-wrapper factorisation for unfolds without ever writing the program in question as an unfold. This is a compelling argument for the flexibility of the theory, but we should also note that because of our choice of assumption and condition none of the extra structure of unfolds is needed, as the equality chain

$$\begin{array}{ll}
 & \mathit{init}' \circ \mathit{rep} \\
= & \{\ (2\beta)\ \} \\
 & \mathit{init} \circ \mathit{abs} \circ \mathit{rep} \\
= & \{\ (\text{C'})\ \} \\
 & \mathit{init}
\end{array}$$

holds regardless of whether init and init' are unfolds. In a sense, because we have chosen the weakest properties, the theory gives us less. This suggests that we should generally prefer to use stronger properties if possible. The use of partially-correct reasoning is also unsatisfactory, as it requires us to appeal to totality.

Using Condition (1)

If we write init explicitly as an unfold, this example can also use condition (1). The unfold for (possibly finite) lists is:

$$\begin{array}{l}
\text{unfold} :: (a \to \mathit{Maybe}\ (b, a)) \to a \to [b] \\
\text{unfold}\ f\ x = \mathbf{case}\ f\ x\ \mathbf{of}
\end{array}$$

```
    Nothing      → []
    Just (b, x') → b : unfold f x'
```

Using this, we can define *init* as follows:

```
init :: [a] → [a]
init = unfold f
  where
    f :: [a] → Maybe (a, [a])
    f []       = ⊥
    f [x]      = Nothing
    f (x : xs) = Just (x, xs)
```

In order to construct a function g such that $init = \text{unfold}\ g \circ rep$, where *rep* is defined as before, we use worker-wrapper factorisation. Condition (1) gives us the explicit definition $g = \mathsf{F}\ rep \circ f \circ abs$. Instantiating this for our particular F, we have:

```
g x = case f (abs x) of
  Nothing      → Nothing
  Just (y, x') → Just (y, rep x')
```

We attempt to simplify this, noting that the type of x is $(a, [a])$. We calculate g separately for the input $(a, [\,])$

```
    g (a, [])
=     { definition of g }
    case f (abs (a, [])) of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { definition of abs }
    case f [a] of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { f [a] = Nothing, cases }
    Nothing
```

and for (a, as), where as is non-empty:

```
    g (a, as)
=     { definition of g }
    case f (abs (a, as)) of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { definition of abs }
    case f (a : as) of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { f (a : as) = Just (a, as) when as nonempty, cases }
    Just (a, rep as)
=     { as is nonempty, let as = a' : as' }
    Just (a, rep (a' : as'))
=     { definition of rep }
    Just (a, (a', as'))
```

We thus obtain the following definition of g:

```
g (a, [])      = Nothing
g (a, a' : as) = Just (a, (a', as))
```

Because we are working in **CPO**, we must also consider the behaviour of the function g on the input $(a, \bot)$:

```
    g (a, ⊥)
=     { specification of g }
    case f (abs (a, ⊥)) of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { definition of abs }
    case f (a : ⊥) of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { f cannot pattern match on a : ⊥ }
    case ⊥ of
      Nothing      → Nothing
      Just (y, x') → Just (y, rep x')
=     { case exhaustion }
    ⊥
```

However, because g is defined by pattern matching on the second component of the tuple, the equation $g\ (a, \bot) = \bot$ clearly holds. Therefore, the above definition of g satisfies worker-wrapper condition (1) and so the factorisation

$$init = \text{unfold}\ g \circ rep$$

is correct. Note that unfold g is precisely the $init'$ function that we derived above, now defined as an unfold.

While we had to write *init* explicitly as an unfold to perform this derivation, the calculation of the improved program was more straightforward than before, and largely mechanical.

Remarks

This above example shows that the same optimised function can sometimes be obtained using different approaches. It is worth noting that this particular optimisation is also an instance of call-pattern specialisation [23], as implemented in the Glasgow Haskell Compiler. However, neither one of these approaches subsumes the other, it simply happens in this case that they coincide.

6.4 Example: Flattening a Tree

Our final example concerns the left-to-right traversal of a binary tree. The naïve way to implement such a traversal is as follows, which corresponds to expressing the function as a fold:

```
data Tree a = Null | Fork (Tree a) a (Tree a)

flatten :: Tree a → [a]
flatten Null           = []
flatten (Fork t1 x t2) = flatten t1 ++ [x] ++ flatten t2
```

However, this approach takes time quadratic in the number of nodes. By fusing this definition with $id = \text{unfold}\ out$, we can transform the function into an unfold. Here is the fusion condition:

```
(out ∘ flatten) t = case g t of
                      Nothing      → Nothing
                      Just (x, t') → Just (x, flatten t')
```

By writing a function *removemin* satisfying the specification of g, we obtain the following new definition of *flatten*:

```
flatten :: Tree a → [a]
flatten = unfold removemin
  where removemin Null = Nothing
        removemin (Fork t1 x t2) =
          case removemin t1 of
            Nothing       → Just (x, t2)
            Just (y, t1') → Just (y, Fork t1' x t2)
```

This approach takes time proportional to $n * l$, where n is the number of nodes and l is the leftwards depth of the tree, i.e. the depth of the deepest node counting only left branches.

However, we can use our worker-wrapper theory to improve this further, exploiting the isomorphism between lists of rose trees (trees with an arbitrary number of children for each node) and binary trees. First we define the type of rose trees:

```
data RoseTree a = RoseTree a [RoseTree a]
```

Now we define a version of flatten for lists of rose trees, in which the list acts as a priority queue of the elements in the original tree, so that at each stage we remove the root of the first tree in the queue, and push all its children onto the front of the queue:

```
flatten' :: [RoseTree a] → [a]
flatten' = unfold g
  where g [] = Nothing
        g (RoseTree x ts1 : ts2) = Just (x, ts1 ++ ts2)
```

We define rep and abs functions to convert between this priority queue representation and the original binary tree representation.

```
rep :: Tree a → [RoseTree a]
rep = reverse ∘ listify
  where listify Null = []
        listify (Fork t1 x t2) =
          RoseTree x (rep t2) : listify t1

abs :: [RoseTree a] → Tree a
abs = delistify ∘ reverse
  where delistify [] = Null
        delistify (RoseTree x ts1 : ts2) =
          Fork (delistify ts2) x (abs ts1)
```

Essentially, rep pulls apart a tree along its left branch into a list, while abs puts the tree back together. The use of $reverse$ is necessary to ensure that the leftmost node is at the head of the list.

The result is an alternate definition $flatten = flatten' \circ rep$ that has comparable performance on balanced trees, but much better performance on trees with long left branches.

We now verify the correctness of this new definition using our worker-wrapper theorem. Firstly, we show that assumption (A) holds, i.e. $abs \circ rep = id$. To do this we note that because $reverse$ is self-inverse, $abs \circ rep = delistify \circ listify$. We prove $delistify \circ listify = id$ by induction on trees. First, the base case:

```
  delistify (listify Null)
=   { definition of listify }
  delistify []
=   { definition of delistify }
  Null
```

Then the inductive case:

```
  delistify (listify (Fork t1 x t2))
=   { definition of listify }
  delistify (RoseTree x (rep t2) : listify t1)
=   { definition of delistify }
  Fork (delistify (listify t1)) x (abs (rep t2))
=   { abs ∘ rep = delistify ∘ listify }
  Fork (delistify (listify t1)) x (delistify (listify t2))
=   { inductive hypothesis }
  Fork t1 x t2
```

Now we must prove that g satisfies one of the worker-wrapper specifications. We choose condition (2) to verify, in this case:

```
removemin (abs ts) =
  case g ts of
    Nothing      → Nothing
    Just (x, ts') → Just (x, abs ts')
```

We verify this equation by induction on the length of the priority queue. For the base case when the queue is empty, we have:

```
  removemin (abs [])
=   { definition of abs }
  removemin Null
=   { definition of removemin }
  Nothing
=   { g [] = Nothing }
  case g [] of
    Nothing      → Nothing
    Just (x, ts') → Just (x, abs ts')
```

For the inductive case, rather than $RoseTree\ x\ ts1 : ts2$ we use $ts1 +\!\!+ [RoseTree\ x\ ts2]$, as abs reverses its argument:

```
  removemin (abs (ts1 ++ [RoseTree x ts2]))
=   { definition of abs }
  removemin (delistify
    (RoseTree x ts2 : reverse ts1))
=   { definition of delistify }
  removemin (Fork (delistify (reverse ts1))
                  x
                  (abs ts2))
=   { abs = delistify ∘ reverse }
  removemin (Fork (abs ts1) x (abs ts2))
=   { definition of removemin }
  case removemin (abs ts1) of
    Nothing      → Just (x, abs ts2)
    Just (y, t1') →
      Just (y, Fork t1' x (abs ts2))
=   { inductive hypothesis }
  case
    case (g ts1) of
      Nothing       → Nothing
      Just (y, ts1') → Just (y, abs ts1')
  of
    Nothing      → Just (x, abs ts2)
    Just (y, t1') →
      Just (y, Fork t1' x (abs ts2))
=   { case of case, pattern matching }
  case g ts1 of
    Nothing       → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, Fork (abs ts1') x (abs ts2))
=   { definition of abs }
  case g ts1 of
    Nothing       → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, Fork (delistify (reverse ts1'))
                    x
                    (abs ts2))
=   { definition of abs }
  case g ts1 of
    Nothing       → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, abs (ts1' ++ [RoseTree x ts2]))
```

We must prove that this last expression is equal to

```
case g (ts1 ++ [RoseTree x ts2]) of
  Nothing      → Nothing
  Just (y, ts') → Just (y, abs ts')
```

which we do by case analysis on ts1. When $ts1 = []$, we have:

```
  case g [] of
    Nothing       → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, abs (ts1' ++ [RoseTree x ts2]))
=   { definition of g, case }
```

$$\begin{array}{ll}
 & \mathit{Just}\ (x, \mathit{abs}\ \mathit{ts2}) \\
= & \{\ \text{case}\ \} \\
 & \textbf{case}\ \mathit{Just}\ (x, \mathit{ts2})\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, \mathit{ts}') \ \rightarrow \mathit{Just}\ (y, \mathit{abs}\ \mathit{ts}') \\
= & \{\ \text{definition of } g\ \} \\
 & \textbf{case}\ g\ [\mathit{RoseTree}\ x\ \mathit{ts2}]\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, \mathit{ts}') \ \rightarrow \mathit{Just}\ (y, \mathit{abs}\ \mathit{ts}') \\
= & \{\ [\,]\ \text{identity of } \mathbin{+\!\!+}\ \} \\
 & \textbf{case}\ g\ ([\,] \mathbin{+\!\!+} [\mathit{RoseTree}\ x\ \mathit{ts2}])\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, \mathit{ts}') \ \rightarrow \mathit{Just}\ (y, \mathit{abs}\ \mathit{ts}')
\end{array}$$

In turn, when $\mathit{ts1} = \mathit{RoseTree}\ z\ \mathit{ts3} : \mathit{ts4}$:

$$\begin{array}{ll}
 & \textbf{case}\ g\ (\mathit{RoseTree}\ z\ \mathit{ts3} : \mathit{ts4})\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Just}\ (x, \mathit{abs}\ \mathit{ts2}) \\
 & \quad \mathit{Just}\ (y, \mathit{ts1}') \rightarrow \\
 & \quad\quad \mathit{Just}\ (y, \mathit{abs}\ (\mathit{ts1}' \mathbin{+\!\!+} [\mathit{RoseTree}\ x\ \mathit{ts2}])) \\
= & \{\ \text{definition of } g \text{, case}\ \} \\
 & \mathit{Just}\ (z, \mathit{abs}\ (\mathit{ts3} \mathbin{+\!\!+} \mathit{ts4} \mathbin{+\!\!+} [\mathit{RoseTree}\ x\ \mathit{ts2}])) \\
= & \{\ \text{case}\ \} \\
 & \textbf{case}\ \mathit{Just}\ (z, \mathit{ts3} \mathbin{+\!\!+} \mathit{ts4} \mathbin{+\!\!+} [\mathit{RoseTree}\ x\ \mathit{ts2}])\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, \mathit{ts}') \ \rightarrow \mathit{Just}\ (y, \mathit{abs}\ \mathit{ts}') \\
= & \{\ \text{definition of } g\ \} \\
 & \textbf{case}\ g\ (\mathit{RoseTree}\ z\ \mathit{ts3} : (\mathit{ts4} \mathbin{+\!\!+} [\mathit{RoseTree}\ x\ \mathit{ts2}]))\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad\quad\ \rightarrow \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, \mathit{ts}') \ \rightarrow \mathit{Just}\ (y, \mathit{abs}\ \mathit{ts}')
\end{array}$$

Therefore, condition (2) and assumption (A) are satisfied, and hence the following worker-wrapper factorisation is valid:

$$\mathit{flatten} = \mathit{flatten}' \circ \mathit{rep}$$

We conclude by noting that while this result can be obtained from fusion alone, the necessary proof is very involved, requiring a lemma about the relationship between $\mathit{removemin}$ and abs. The worker-wrapper proof, while long, is mechanical. For comparison, the fusion-based proof is available on the web at `http://www.cs.nott.ac.uk/~jph/flatten_fusion.pdf`.

7. Related Work

We have divided the related work into four categories. The first two relate to the history of the unfold operator in programming languages and category theory respectively. The third relates to the use of fusion in program optimisation, while the fourth relates to applications of program factorisation.

7.1 Unfold in Programming Languages

The use of unfold in programming is a lot more recent than that of fold. While fold-like operations trace their history back to APL [16], the earliest unfold-like mechanism appears to be in Miranda list comprehensions [27], which have a special ". ." syntax that can be used to express unfold-like computations without the need for explicit recursion. However, in Miranda there was no dedicated unfold operator such as the one in Haskell, which became part of the standard library in Haskell 98 [22].

The unfold operator seems to first appear in a recognisable form in 1988 in *Introduction to Functional Programming* by Bird and Wadler [1], where it is defined in terms of map, $\mathit{takeWhile}$ and $\mathit{iterate}$. No direct recursive definition is given, and it only appears on a single page. Meijer, Fokkinga and Paterson noted in 1991 that the unfold operator from [1] was categorically dual to fold [18].

In 1998, Gibbons and Jones published the paper *The Under-Appreciated Unfold* [9], which gave the inspiration for the title of this paper. The paper argued that unfold was an underutilised programming tool, and justified this by presenting algorithms for breadth-first traversal using both fold and unfold, arguing that the unfold-based algorithms were clearer.

7.2 Unfold in Category Theory

It seems that categorical unfolds (also known as anamorphisms) were largely developed in parallel to unfold in programming languages. The earliest mention of categorical unfold appears to be in Hagino's 1987 PhD thesis *A Categorical Programming Language* [13] and subsequently in Malcolm's 1990 thesis *Algebraic Data Types and Program Transformation* [17]. Interestingly, neither of these make any linguistic distinction between folds and unfolds, using the same terminology to refer to both. It seems that this distinction was not made until Meijer et al.'s 1991 paper *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* [18], which concerned folds and unfolds in $\mathbf{CPO}$ and essentially unified the programming language work with the categorical work.

7.3 Program Fusion

In functional programming, many successful program optimisations are based upon fusion, in which separate parts of a program are fused together to eliminate intermediate data structures. Fusion was first introduced by Wadler in 1990 by the name of *deforestation* [31]. Since then, it has been widely explored, especially in regards to specific recursion patterns.

A particularly successful example is *foldr/build* fusion, in which list-producers are defined in terms of a function build while list-consumers are defined in terms of foldr. Introduced by Gill, Launchbury and Peyton Jones [11], this pattern has been the subject of much research [2, 26, 30] and is implemented in GHC.

A more recent innovation by Coutts, Leshchinsky and Stewart is *stream fusion*, a way to optimise list-processing functions by changing the representation of lists [4]. This approach makes essential use of an unfold-like operator, and is related to worker-wrapper factorisation as it involves a change of intermediate data type. However, stream fusion utilises the recursive structure of the underlying data, whereas our approach does not.

7.4 Program Factorisation

Compared to fusion-based techniques, program factorisation or "fission" seems far less well-explored. Gibbons' 2006 paper, *Fission for Program Comprehension* [7], presents an application of this idea which differs from our work in two ways. Firstly, Gibbons does not concern himself with program optimisation; rather, his intended use is understanding an already-written program by breaking it down into the combination of separate parts that are easier to comprehend. Secondly, while Gibbons' approach is based upon applying fusion in reverse, the proof of our approach to program factorisation actually involves a forward application of fusion.

8. Conclusion

In this paper, we have presented a novel approach to optimising programs written in the form of an unfold, showing how a useful approach comes from the commonly-used idea of generalising a coinductive hypothesis. We have provided a general factorisation theorem that can be used either to guide the derivation of an optimised program or to prove such a program correct, resulting in a technique that we believe has wide applicability. We demonstrated the utility of our technique with a collection of examples.

8.1 Further Work

We have only considered programs in the form of an unfold, but there are other corecursive patterns that can be considered. One example is *apomorphisms* [29], which capture the idea of *primitive corecursion*, allowing construction of the result to short-cut by producing the remainder of the result in a single step. The apomorphism operator for streams can be defined as follows:

$$\begin{array}{ll} \mathit{apo}\ h\ t\ x = h\ x\ :\ \mathbf{case}\ t\ x\ \mathbf{of} & \\ \qquad\qquad\qquad\qquad \mathit{Left}\ x' & \rightarrow \mathit{apo}\ h\ t\ x' \\ \qquad\qquad\qquad\qquad \mathit{Right}\ \mathit{xs} & \rightarrow \mathit{xs} \end{array}$$

Apomorphisms have their own fusion law, so it seems likely that they would have a useful worker-wrapper factorisation theorem.

Another possible direction is to adapt our method to deal with circular definitions. For example, we can write a circular definition of the infinite stream of Fibonacci numbers:

$$\mathit{fibs} = 0\ :\ 1\ :\ \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})$$

Coinductive techniques can be used to reason about circular definitions like this one, but whether a factorisation theorem analogous to ours exists for such definitions remains to be seen.

We could also consider extending this work to monadic and comonadic unfolds. Monadic unfolds [21] are of particular interest; consider the monadic unfold operator for streams:

$$\begin{array}{l} \mathrm{unfoldM} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ (b, a)) \rightarrow \\ \qquad\qquad\qquad\qquad\qquad a \rightarrow m\ (\mathit{Stream}\ b) \\ \mathrm{unfoldM}\ f\ x = \mathbf{do}\ (b, x') \leftarrow f\ x \\ \qquad\qquad\qquad\qquad\ \ \mathit{bs} \leftarrow \mathrm{unfoldM}\ f\ x' \\ \qquad\qquad\qquad\qquad\ \ \mathit{return}\ (b\ :\ \mathit{bs}) \end{array}$$

Any monad with a strict bind operator will fail to produce a result, instead simply bottoming out. Intuitively, the problem is that an infinite number of effects must be applied before the final result can be produced. Thus we see that there is a fundamental difference between ordinary and monadic unfolds that raises interesting questions concerning the worker-wrapper theory.

The theory we have presented is concerned with correctness. In order to reason about efficiency gains, we also need an operational theory, for which purposes we are currently exploring the use of improvement theory [19]. Finally, while we have noted that the proofs are often mechanical, we have yet to consider how our new theory may be *mechanised*. A team at the University of Kansas is currently working on the implementation of various worker-wrapper theories as an extension to the Glasgow Haskell Compiler [5, 25], with promising initial results.

Acknowledgments

The authors would like to thank Richard Bird for assistance with tracing the history of the unfold operator. We would also like to thank Philippa Cowderoy for the observation in section 6.4 that the unfold-based *flatten* could be derived from the fold-based *flatten*.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. 1988.

[2] O. Chitil. Type Inference Builds a Short Cut to Deforestation. In *ICFP '99*. ACM, 1999.

[3] T. Coquand. Infinite Objects in Type Theory. In *TYPES '93*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1993.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *ICFP '07*. ACM, 2007.

[5] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell Symposium (Haskell '12)*. ACM, 2012.

[6] P. J. Freyd. Remarks on Algebraically Compact Categories. In *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1992.

[7] J. Gibbons. Fission for Program Comprehension. In *MPC '06*, volume 4014 of *Lecture Notes in Computer Science*. Springer, 2006.

[8] J. Gibbons and G. Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4), April-May 2005.

[9] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *ICFP '98*. ACM, 1998.

[10] A. Gill and G. Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2), Mar. 2009.

[11] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *FPCA '93*. Springer, 1993.

[12] A. D. Gordon. A Tutorial on Co-induction and Functional Programming. In *In Glasgow Functional Programming Workshop*. Springer, 1994.

[13] T. Hagino. *A Categorical Programming Language*. PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

[14] R. Hinze. Functional Pearl: Streams and Unique Fixed Points. In *ICFP '08*, New York, NY, USA, 2008.

[15] G. Hutton, M. Jaskelioff, and A. Gill. Factorising Folds for Faster Functions. *Journal of Functional Programming Special Issue on Generic Programming*, 20(3&4), June 2010.

[16] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[17] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, 1990.

[18] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA '91*, volume 523 of *Lecture Notes in Computer Science*. Springer, 1991.

[19] A. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL '99*, pages 43–56. ACM, 1999.

[20] L. S. Moss and N. Danner. On the Foundations of Corecursion. *Logic Journal of the IGPL*, 5(2), 1997.

[21] A. Pardo. Monadic Corecursion — Definition, Fusion Laws, and Applications. *Electr. Notes Theor. Comput. Sci.*, 11, 1998.

[22] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[23] S. Peyton Jones. Call-Pattern Specialisation for Haskell Programs. In *ICFP '07*. ACM, 2007.

[24] N. Sculthorpe and G. Hutton. Work It, Wrap It, Fix It, Fold It. Submitted to the Journal of Functional Programming, 2013.

[25] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Draft Proceedings of Implementation and Application of Functional Languages (IFL '12)*, 2012.

[26] A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *FPCA '95*. Springer, 1995.

[27] D. A. Turner. *Miranda System Manual*. Research Software Ltd., Canterbury, England, 1989. Available online at http://miranda.org.uk/.

[28] D. A. Turner. Elementary Strong Functional Programming. In *FPLE '95*, volume 1022 of *Lecture Notes in Computer Science*. Springer, 1995.

[29] T. Uustalu and V. Vene. Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically. *Informatica, Lith. Acad. Sci.*, 10 (1), 1999.

[30] J. Voigtländer. Proving Correctness via Free Theorems: The Case of the destroy/build-Rule. In *PEPM '08*. ACM, 2008.

[31] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2), 1990.