Electronic Notes in Theoretical Computer Science 44 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 15 pages

When is a function a fold or an unfold?

Jeremy Gibbons

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

Graham Hutton

Languages and Programming Group, School of Computer Science and IT, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom

Thorsten Altenkirch

Languages and Programming Group, School of Computer Science and IT, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom

Abstract

We give a necessary and sufficient condition for when a set-theoretic function can be written using the recursion operator fold, and a dual condition for the recursion operator unfold. The conditions are simple, practically useful, and generic in the underlying datatype.

1 Introduction

The recursion operator fold encapsulates a common pattern for defining programs that *consume* values of a *least* fixpoint type such as finite lists. Dually, the recursion operator unfold encapsulates a common pattern for defining programs that *produce* values of a *greatest* fixpoint type such as streams (infinite lists). Theory and applications of fold abound — see [11,4] for recent surveys — while in recent years it has become increasingly clear that the less well-known concept of unfold is just as useful [5,6,10,13,15].

Given the interest in fold and unfold, it is natural to ask when a program can be written using one of these operators. Surprisingly little is known about this question. This article gives a complete answer for the special case in which programs are total functions between sets. In particular, we give a necessary and sufficient condition for when a set-theoretic function can be written using fold, and a dual condition for unfold. The conditions are simple, practically useful, and generic in the underlying datatype. However, our proofs are set-theoretic, and make essential use of classical logic and the Axiom of Choice; hence our results do not generalize to categories of constructive functions[1].

2 Fold and unfold

In this section we review the categorical treatment of fold and unfold in terms of initial algebras and final coalgebras; for further details see [18,20,14,1].

Suppose that we fix a category $\mathcal{C}$ and a functor $F : \mathcal{C} \to \mathcal{C}$. An *algebra* is pair (A, f) comprising an object A and an arrow $f : F\,A \to A$, and a *homomorphism* $h : (A, f) \to (B, g)$ from one such algebra to another is an arrow $h : A \to B$ such that the following square commutes:

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

An *initial algebra* is an initial object in the category with algebras as objects and homomorphisms as arrows. We write $(\mu F, \mathsf{in})$ for an initial algebra, and $\mathsf{fold}\,f$ for the unique homomorphism $h : (\mu F, \mathsf{in}) \to (A, f)$ from the initial algebra to any other algebra (A, f). That is, $\mathsf{fold}\,f$ is defined as the unique arrow that makes the following square commute:

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

The dual notions of *coalgebra*, *cohomomorphism*, and *terminal coalgebra* are defined similarly. We write $(\nu F, \mathsf{out})$ for a terminal coalgebra, and $\mathsf{unfold}\,f$ for the unique cohomomorphism $h : (A, f) \to (\nu F, \mathsf{out})$ from any coalgebra (A, f) to the terminal coalgebra. That is, $\mathsf{unfold}\,f$ is defined as the unique arrow that makes the following square commute:

$$\begin{tikzcd} A \arrow[r, dotted, "\mathsf{unfold}\,f"] \arrow[d, "f"'] & \nu F \arrow[d, "\mathsf{out}"] \\ F\,A \arrow[r, "F\,(\mathsf{unfold}\,f)"'] & F\,(\nu F) \end{tikzcd}$$

In the literature, $\mathsf{fold}\,f$ and $\mathsf{unfold}\,f$ are sometimes written as $(\!| f |\!)$ and $[\![f]\!]$,

[1] Such as the effective topos or the category of ω-sets.

and called *catamorphisms* and *anamorphisms* respectively.

2.1 Example: finite lists

Suppose that we define a functor $L : \mathcal{SET} \to \mathcal{SET}$ by $L\,A = \mathbf{1}+(\mathbb{N}\times A)$ and $L\,f = \mathsf{id}_{\mathbf{1}}+(\mathsf{id}_{\mathbb{N}}\times f)$, where $\mathbb{N}$ is the set of natural numbers. Then an algebra is a pair (A, f) comprising a set A and a function $f : \mathbf{1}+(\mathbb{N}\times A) \to A$. Functions of this type can always be uniquely decomposed into the form $f = [g, h]$ for some other functions $g : \mathbf{1} \to A$ and $h : \mathbb{N} \times A \to A$. A homomorphism $f : (A, [g, h]) \to (B, [i, j])$ is a function $f : A \to B$ such that $f \cdot g = i$ and $f \cdot h = j \cdot (\mathsf{id}_{\mathbb{N}} \times f)$.

The functor L has an initial algebra $(\mu L, \mathsf{in}) = (\mathit{List}(\mathbb{N}), [\mathit{nil}, \mathit{cons}])$, where $\mathit{List}(A)$ is the set of all finite lists with elements drawn from A, and $\mathit{nil} : \mathbf{1} \to \mathit{List}(\mathbb{N})$ and $\mathit{cons} : \mathbb{N} \times \mathit{List}(\mathbb{N}) \to \mathit{List}(\mathbb{N})$ are constructors for this set. Given any other set A and two functions $i : \mathbf{1} \to A$ and $j : \mathbb{N} \times A \to A$, the function $\mathsf{fold}\,[i, j] : \mathit{List}(\mathbb{N}) \to A$ is uniquely defined by the following two equations:

$$
\begin{aligned}
\mathsf{fold}\,[i, j] \cdot \mathit{nil} &= i \\
\mathsf{fold}\,[i, j] \cdot \mathit{cons} &= j \cdot (\mathsf{id}_{\mathbb{N}} \times \mathsf{fold}\,[i, j])
\end{aligned}
$$

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

We will use this datatype in examples later. For notational simplicity, we will write '$[\,]$' for $\mathit{nil}\,()$, and '$x : \mathit{xs}$' for $\mathit{cons}\,(x, \mathit{xs})$. Thus, we might have written the above definition of fold more perspicuously as:

$$
\begin{aligned}
(\mathsf{fold}\,[i, j])\,[\,] &= i \\
(\mathsf{fold}\,[i, j])\,(x : \mathit{xs}) &= j\,(x, (\mathsf{fold}\,[i, j])\,\mathit{xs})
\end{aligned}
$$

2.2 Example: streams

Suppose that we define a functor $S : \mathcal{SET} \to \mathcal{SET}$ by $SA = \mathbb{N} \times A$ and $Sf = \mathsf{id}_{\mathbb{N}} \times f$. Then a coalgebra is a pair (A, f) comprising a set A and a function $f : A \to \mathbb{N} \times A$. Functions of this type can always be uniquely decomposed into the form $f = \langle g, h\rangle$ for some other functions $g : A \to \mathbb{N}$ and $h : A \to A$. A cohomomorphism $f : (A, \langle g, h\rangle) \to (B, \langle i, j\rangle)$ is a function $f : A \to B$ such that $i \cdot f = g$ and $j \cdot f = f \cdot h$.

The functor S has a terminal coalgebra $(\nu S, \mathsf{out}) = (\mathit{Stream}(\mathbb{N}), \langle \mathit{head}, \mathit{tail}\rangle)$, where $\mathit{Stream}(A)$ is the set of all streams with elements drawn from A, and $\mathit{head} : \mathit{Stream}(\mathbb{N}) \to \mathbb{N}$ and $\mathit{tail} : \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N})$ are destructors for this set. Given any other set A and two functions $g : A \to \mathbb{N}$ and $h : A \to A$, the function $\mathsf{unfold}\,\langle g, h\rangle : A \to \mathit{Stream}(\mathbb{N})$ is uniquely defined by the following two equations:

$$\begin{array}{lcl} head \cdot \mathsf{unfold}\,\langle g,h\rangle & = & g \\ tail \cdot \mathsf{unfold}\,\langle g,h\rangle & = & \mathsf{unfold}\,\langle g,h\rangle \cdot h \end{array}$$

That is, $\mathsf{unfold}\,\langle g,h\rangle$ produces a stream by using the function g to produce the *head* of the stream, and the function h to generate another value that is then itself unfolded in the same way to produce the *tail* of the stream. For example, the function $from : \mathbb{N} \to Stream(\mathbb{N})$, which produces a stream of naturals ascending in steps of one, can be defined by $from = \mathsf{unfold}\,\langle \mathsf{id}_{\mathbb{N}}, succ\rangle$ where $succ : \mathbb{N} \to \mathbb{N}$ is given by $succ\ x = x + 1$.

3 When is an arrow a fold or an unfold?

The fold operator encapsulates a common pattern for defining an arrow of type $\mu F \to A$. It is natural then to ask when an arrow of this type can be written using fold. More precisely, when can an arbitrary arrow $h : \mu F \to A$ be written in the form $h = \mathsf{fold}\,f$ for some other arrow $f : F A \to A$?

A technically complete, but nonetheless unsatisfactory, answer to this question is provided by the universal property of the fold operator [18], which can be stated as the following equivalence:

$$h = \mathsf{fold}\,f \quad\Leftrightarrow\quad h \cdot \mathsf{in} = f \cdot F\,h$$

The $\Rightarrow$ direction of this equivalence states that $\mathsf{fold}\,f$ is a homomorphism from the initial algebra $(\mu F, \mathsf{in})$ to another algebra (A, f), while the $\Leftarrow$ direction states that any other homomorphism h between these two algebras must be equal to $\mathsf{fold}\,f$. Taken as a whole, the universal property expresses the fact that $\mathsf{fold}\,f$ is the *unique* homomorphism from $(\mu F, \mathsf{in})$ to (A, f).

The universal property provides a complete answer to our question — h can be written in the form $\mathsf{fold}\,f$ precisely when $h \cdot \mathsf{in} = f \cdot F\,h$ — but is less helpful than it might be because it requires that we already know f. Given a specific h, however, the universal property can often be used to guide the construction of an appropriate f [11], but we do not consider this a completely satisfactory answer either, because this approach is only a heuristic, and it is sometimes difficult to apply in practice.

The problem with the universal property is that it concerns an *intensional* aspect of h, namely the function f that forms part of its implementation. Often a condition based on purely *extensional* aspects is more useful. A partial answer to our question with purely extensional concerns is that every left invertible arrow $h : \mu F \to A$ can be written using fold [20]. Formally, if we assume that there exists an arrow $g : A \to \mu F$ such that $g \cdot h = \mathsf{id}_{\mu F}$, then the equation $h = \mathsf{fold}\,f$ can be solved for f as follows:

$$
\begin{array}{ll}
& h \;=\; \mathsf{fold}\, f \\
\Leftrightarrow & \quad \{ \text{ universal property } \} \\
& h \cdot \mathsf{in} \;=\; f \cdot F\, h \\
\Leftrightarrow & \quad \{ \text{ identities } \} \\
& h \cdot \mathsf{in} \cdot \mathsf{id}_{F\,(\mu F)} \;=\; f \cdot F\, h \\
\Leftrightarrow & \quad \{ \text{ functors } \} \\
& h \cdot \mathsf{in} \cdot F\,(\mathsf{id}_{\mu F}) \;=\; f \cdot F\, h \\
\Leftrightarrow & \quad \{ \text{ assumption } \} \\
& h \cdot \mathsf{in} \cdot F\,(g \cdot h) \;=\; f \cdot F\, h \\
\Leftrightarrow & \quad \{ \text{ functors } \} \\
& h \cdot \mathsf{in} \cdot F\, g \cdot F\, h \;=\; f \cdot F\, h \\
\Leftarrow & \quad \{ \text{ substitutivity } \} \\
& f \;=\; h \cdot \mathsf{in} \cdot F\, g
\end{array}
$$

In summary, we have derived the following implication:

$$g \cdot h \;=\; \mathsf{id}_{\mu F} \quad \Rightarrow \quad h \;=\; \mathsf{fold}\,(h \cdot \mathsf{in} \cdot F\, g)$$

As an example, the function $\mathit{rev} : \mathit{List}(\mathbb{N}) \to \mathit{List}(\mathbb{N})$ that reverses a list is its own inverse, and hence it is immediate that rev can be written using fold by the above implication. Note, however, that this implication only provides a partial answer to our question, because the converse is not true in general. That is, not every arrow $h : \mu F \to A$ that can be written using fold is left invertible. For example, the function $\mathit{sum} : \mathit{List}(\mathbb{N}) \to \mathbb{N}$ was written using fold in the previous section, but is not left invertible.

Dually, the unfold operator also satisfies a universal property, which can be used to show that every right invertible arrow of type $A \to \nu F$ can be written using unfold [20]. For example, the function $\mathit{evenpos} : \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N})$ that removes every other element from a stream has a right inverse (any function that inserts an element between each adjacent pair in a stream), and hence it is immediate that $\mathit{evenpos}$ can be written using unfold. However, not every arrow $h : A \to \nu F$ that can be written using unfold is right invertible. For example, the function $\mathit{from} : \mathbb{N} \to \mathit{Stream}(\mathbb{N})$ was written using unfold in the previous section, but is not right invertible.

As far as we are aware, the invertibility results above are the only known results that state when arbitrary arrows of the correct type can be written using fold or unfold. We conclude this section by noting that much more progress has been made concerning specific kinds of arrows. For example, the fusion law states that the composition of a homomorphism and a fold can always be written as a fold, while the banana split law states that two folds applied to the same argument can always be written as a single fold [20].

4 When is a function a fold?

In this section we give a necessary and sufficient condition for when an arrow can be written using fold, for the special case of the category $\mathcal{S}ET$ in which the arrows are total functions between sets. We dualize the result to unfold in the following section.

The result depends on the following definition:

Definition 4.1 The *kernel* [17] of a function $f : A \to B$ is the set of pairs of elements that are identified by f:

$$\mathsf{ker}\, f \quad = \quad \{\ (a, a') \in A \times A \mid f\, a = f\, a'\ \}$$

The main result of this section is a necessary and sufficient condition for when an arbitrary arrow $h : \mu F \to A$ in $\mathcal{S}ET$ can be written in the form $h = \mathsf{fold}\, f$ for some other arrow $f : F\, A \to A$.

Theorem 4.2 *Suppose that $h : \mu F \to A$. Then*

$$(\exists g : F\, A \to A. \quad h = \mathsf{fold}\, g) \quad \Leftrightarrow \quad \mathsf{ker}\,(F\, h) \subseteq \mathsf{ker}\,(h \cdot \mathsf{in})$$

(Another way of saying this is that h is a fold iff $\mathsf{ker}\, h$ is a congruence under in; that is, writing $\mathrm{Rel}(F)(R)$ for the *relational lifting* to a relation on $F\, A$ of relation R on A [12], iff $(x, y) \in \mathrm{Rel}(F)(\mathsf{ker}\, h)$ implies $(\mathsf{in}\, x, \mathsf{in}\, y) \in \mathsf{ker}\, h$.)

The crux of the proof is the well-known observation that inclusion of kernels is equivalent to the existence of 'postfactors':

Lemma 4.3 *Suppose that $f : A \to B$ and $h : A \to C$. Then*

$$(\exists g : B \to C. \quad h = g \cdot f) \quad \Leftrightarrow \quad (\mathsf{ker}\, f \subseteq \mathsf{ker}\, h \ \wedge\ B \to C \neq \emptyset)$$

Proof. The proof is straightforward. For the $\Rightarrow$ direction, assume that $g : B \to C$ and $h = g \cdot f$; then clearly $B \to C \neq \emptyset$, and moreover,

$$
\begin{array}{ll}
& (a, a') \in \mathsf{ker}\, f \\
\Leftrightarrow & \quad \{ \text{ kernels } \} \\
& f\, a = f\, a' \\
\Rightarrow & \quad \{ \text{ substitutivity } \} \\
& g\,(f\, a) = g\,(f\, a') \\
\Leftrightarrow & \quad \{\ h = g \cdot f\ \} \\
& h\, a = h\, a' \\
\Leftrightarrow & \quad \{ \text{ kernels } \} \\
& (a, a') \in \mathsf{ker}\, h
\end{array}
$$

Conversely, assume that $\mathsf{ker}\, f \subseteq \mathsf{ker}\, h$ and $B \to C \neq \emptyset$, so that either $B = \emptyset$ or $C \neq \emptyset$. When $B = \emptyset$, let g be the unique function in $B \to C$; note that g is the 'empty function', and so $g \cdot f$ is empty too. Moreover, $A = \emptyset$ because of the type of f, so h is also empty and hence equal to $g \cdot f$. When $C \neq \emptyset$, we define $g\, b$ for b in the range of f by $g\, b = h\, a$ for some a with $f\, a = b$; this is a proper definition, because if there are two choices a, a' with $f\, a = f\, a' = b$, then $h\, a = h\, a'$ also by assumption. For b outside the range of f, we define $g\, b$ arbitrarily. By construction, this gives $h\, a = g\,(f\, a)$ for every a. □

We also use the following simple fact concerning initial algebras:

Lemma 4.4

$$\mu F \to A \neq \emptyset \quad \Rightarrow \quad F\, A \to A \neq \emptyset$$

Proof. We note that $F\, A \to A \neq \emptyset$ is equivalent to $A = \emptyset \;\Rightarrow\; F\, A = \emptyset$, which implication can then be verified as follows:

$$
\begin{array}{ll}
& A = \emptyset \\
\Rightarrow & \quad \{\ \mu F \to A \neq \emptyset\ \} \\
& \mu F = \emptyset \\
\Rightarrow & \quad \{\ \mathsf{in} : F\,(\mu F) \to \mu F\ \} \\
& F\,(\mu F) = \emptyset \\
\Rightarrow & \quad \{\ \mu F = \emptyset = A\ \} \\
& F\, A = \emptyset
\end{array}
$$

□

Proof of Theorem 4.2 Given the two lemmata above, the proof of the theorem is almost embarrassingly simple:

$$
\begin{aligned}
& \exists g : F\,A \to A. \quad h = \mathsf{fold}\,g \\
\Leftrightarrow \quad & \quad \{ \text{ universal property } \} \\
& \exists g : F\,A \to A. \quad h \cdot \mathsf{in} = g \cdot F\,h \\
\Leftrightarrow \quad & \quad \{ \text{ Lemma 4.3 } \} \\
& \mathsf{ker}\,(F\,h) \subseteq \mathsf{ker}\,(h \cdot \mathsf{in}) \;\wedge\; F\,A \to A \neq \emptyset \\
\Leftrightarrow \quad & \quad \{ \text{ Lemma 4.4, } h : \mu F \to A \text{ } \} \\
& \mathsf{ker}\,(F\,h) \subseteq \mathsf{ker}\,(h \cdot \mathsf{in})
\end{aligned}
$$

□

Remark 4.5 For the type $List(A)$ of finite lists with elements drawn from A, with constructors $nil : \mathbf{1} \to List(A)$ and $cons : A \times List(A) \to List(A)$, Theorem 4.2 reduces to stating that an arbitrary function $h : List(A) \to B$ can be written directly as a fold precisely when the lists that are identified by h are closed under $cons$, in the sense that for all x, xs, ys,

$$h\,xs \;=\; h\,ys \quad \Rightarrow \quad h\,(x : xs) \;=\; h\,(x : ys)$$

Example 4.6 If we define $\mathit{sum} : \mathit{List}(\mathbb{N}) \to \mathbb{N}$ by the equations

$$\begin{aligned} \mathit{sum}\,[\,] &= 0 \\ \mathit{sum}\,(x : \mathit{xs}) &= x + \mathit{sum}\,\mathit{xs} \end{aligned}$$

then it is easy to show that the lists identified by *sum* are closed under *cons*:

$$\begin{array}{ll} & \mathit{sum}\,(x : \mathit{xs}) \;=\; \mathit{sum}\,(x : \mathit{ys}) \\ \Leftrightarrow & \quad \{ \text{ definition of } \mathit{sum} \ \} \\ & x + \mathit{sum}\,\mathit{xs} \;=\; x + \mathit{sum}\,\mathit{ys} \\ \Leftarrow & \quad \{ \text{ substitutivity } \} \\ & \mathit{sum}\,\mathit{xs} \;=\; \mathit{sum}\,\mathit{ys} \end{array}$$

Hence, *sum* can be written directly using fold.

Example 4.7 In contrast, if we define a function $\mathit{stail} : \mathit{List}(\mathbb{N}) \to \mathit{List}(\mathbb{N})$ (for 'safe tail') by the equations

$$\begin{aligned} \mathit{stail}\,[\,] &= [\,] \\ \mathit{stail}\,(x : \mathit{xs}) &= \mathit{xs} \end{aligned}$$

then a simple counterexample verifies that the lists identified by *stail* are not closed under *cons*: for example, with $\mathit{xs} = [\,]$ and $\mathit{ys} = 0 : [\,]$, we have $\mathit{stail}\,\mathit{xs} = [\,] = \mathit{stail}\,\mathit{ys}$, but $\mathit{stail}\,(1 : \mathit{xs}) = [\,] \neq 0 : [\,] = \mathit{stail}\,(1 : \mathit{ys})$. Therefore *stail* cannot be written directly as a fold.

Example 4.8 For the type $\mathit{List}(\mathbb{R})$ of finite lists of reals, consider the problem of computing $\mathit{floorsum} = \mathit{floor} \cdot \mathit{rsum}$, where $\mathit{rsum} : \mathit{List}(\mathbb{R}) \to \mathbb{R}$ sums a list of reals and $\mathit{floor} : \mathbb{R} \to \mathbb{Z}$ rounds a real r down to the largest integer at most r. Because the result is an integer, one might wonder whether *floorsum* can be carried out as a fold to integers, thereby avoiding the computationally more expensive real arithmetic. It cannot: we have $\mathit{floorsum}\,(0.3 : [\,]) = \mathit{floorsum}\,(0.6 : [\,])$, but $\mathit{floorsum}\,(0.5 : 0.3 : [\,]) \neq \mathit{floorsum}\,(0.5 : 0.6 : [\,])$.

On the other hand, the reverse composition $\mathit{sum} \cdot \mathit{map}\,\mathit{floor}$, which floors every element of the list before summing, can be written as a fold: an argument similar to Example 4.6 applies. This is an instance of *deforestation* [24], an optimisation whereby two computations are combined into one and the intermediate data structure (here of type $\mathit{List}(\mathbb{Z})$) is eliminated.

Remark 4.9 For the type $\mathit{Tree}(A)$ of binary trees with constructors $\mathit{leaf} : A \to \mathit{Tree}(A)$ and $\mathit{node} : \mathit{Tree}(A) \times \mathit{Tree}(A) \to \mathit{Tree}(A)$, Theorem 4.2 reduces to stating that an arbitrary function $h : \mathit{Tree}(A) \to B$ can be written directly as a fold precisely when the trees that are identified by h are closed under *node*, in the sense that for all t, u,

$$h\,t = h\,t' \;\wedge\; h\,u = h\,u' \quad \Rightarrow \quad h\,(\mathit{node}\,(t, u)) = h\,(\mathit{node}\,(t', u'))$$

Example 4.10 For another deforestation example, consider $\mathit{flatsum} = \mathit{sum} \cdot \mathit{flatten}$, where $\mathit{flatten} : \mathit{Tree}(A) \to \mathit{List}(A)$ generates a list of the elements of a tree. The intermediate list in *flatsum* can be eliminated, because

$$
\begin{array}{cl}
& \mathit{flatsum}\,(\mathit{node}\,(t,u)) \\
= & \quad \{ \text{ definition of } \mathit{flatsum} \ \} \\
& \mathit{sum}\,(\mathit{flatten}\,(\mathit{node}\,(t,u))) \\
= & \quad \{ \text{ definition of } \mathit{flatten} \ \} \\
& \mathit{sum}\,(\mathit{flatten}\,t \mathbin{+\!\!+} \mathit{flatten}\,u) \\
= & \quad \{ \ \mathit{sum} \text{ distributes over } \mathbin{+\!\!+} \ \} \\
& \mathit{sum}\,(\mathit{flatten}\,t) + \mathit{sum}\,(\mathit{flatten}\,u) \\
= & \quad \{ \text{ definition of } \mathit{flatsum} \ \} \\
& \mathit{flatsum}\,t + \mathit{flatsum}\,u
\end{array}
$$

from which we conclude that trees identified under *flatsum* are closed under *node*. (Here, '$\mathbin{+\!\!+}$' concatenates two lists.)

Example 4.11 The predicate $\mathit{bal} : \mathit{Tree}(A) \to \mathbb{B}$ that holds of tree iff it is *balanced* (all the leaves at the same depth) is not a fold: with tree t being balanced and of depth 1, and tree u being balanced and of depth 2, both t and u are identified by *bal* (both yielding true), yet $\mathit{bal}\,(\mathit{node}\,(t,t)) \neq \mathit{bal}\,(\mathit{node}\,(t,u))$.

Example 4.12 However, the function $\mathit{dbal} : \mathit{Tree}(A) \to \mathbb{N} \times \mathbb{B}$ that computes a pair, the depth of the tree and whether it is balanced, is a fold. Because

$$
\begin{array}{lcl}
\mathit{depth}\,(\mathit{node}\,(t,u)) & = & 1 + \mathit{max}\,(\mathit{depth}\,t, \mathit{depth}\,u) \\
\mathit{bal}\,(\mathit{node}\,(t,u)) & = & \mathit{bal}\,t \ \wedge \ \mathit{bal}\,u \ \wedge \ \mathit{depth}\,t = \mathit{depth}\,u
\end{array}
$$

trees identified by *dbal* are closed under *node*. This is an example of a *mutumorphism* [7] or *almost homomorphism* [3,8]; transforming a function into such a form is an important step towards constructing an efficient data-parallel algorithm for computing it.

5 When is a function an unfold?

Dualising Theorem 4.2 to unfold is straightforward. The appropriate dual to the notion of the kernel of a function is simply its *image*:

Definition 5.1 The *image* of a function $f : A \to B$ is the set of elements that are produced by f:

$$\mathsf{img}\,f \quad = \quad \{\, b \in B \mid \exists a \in A. \quad f\,a = b \,\}$$

The duality between kernels and images is perhaps not immediately evident, but is revealed by thinking relationally. In particular, if functions are viewed as relations in the obvious way, then the relational composition $f^\circ \cdot f$ of a function f with its converse f° is precisely the kernel of f, while the dual composition $f \cdot f^\circ$ is (the identity relation on) the image of f.

We can now present our result for unfold, which gives a necessary and sufficient condition for when an arbitrary arrow $h : A \to \nu F$ in $\mathcal{S}ET$ can be written in the form $h = \mathsf{unfold}\, g$ for some other arrow $g : A \to F\,A$.

Theorem 5.2 *Suppose that* $h : A \to \nu F$. *Then*

$$(\exists g : A \to F\,A.\quad h = \mathsf{unfold}\, g) \quad\Leftrightarrow\quad \mathsf{img}\,(F\,h) \supseteq \mathsf{img}\,(\mathsf{out} \cdot h)$$

(Another way of saying this is that h is an unfold iff $\mathsf{img}\, h$ is an invariant of out; that is, writing $\mathrm{Pred}(F)(P)$ for the *predicate lifting* to a predicate on $F\,A$ of predicate P on A [12], iff $\mathrm{Pred}(F)(\in \mathsf{img}\, h)\,(\mathsf{out}\, x)$ follows from $(\in \mathsf{img}\, h)\, x$.)

The crux of the proof is the dual of Lemma 4.3, namely that inclusion of images is equivalent to the existence of 'prefactors':

Lemma 5.3 *Suppose that* $f : B \to C$ *and* $h : A \to C$. *Then*

$$(\exists g : A \to B.\quad h = f \cdot g) \quad\Leftrightarrow\quad (\mathsf{img}\, f \supseteq \mathsf{img}\, h \;\wedge\; A \to B \neq \emptyset)$$

Proof. For the $\Rightarrow$ direction, assume that $g : A \to B$ and $h = f \cdot g$; then clearly $A \to B \neq \emptyset$, and moreover,

$$\begin{array}{ll}
& c \in \mathsf{img}\, h \\
\Leftrightarrow & \quad \{\ \text{images}\ \} \\
& \exists a.\quad h\,a = c \\
\Leftrightarrow & \quad \{\ h = f \cdot g\ \} \\
& \exists a.\quad f\,(g\,a) = c \\
\Rightarrow & \quad \{\ g : A \to B\ \} \\
& \exists b.\quad f\,b = c \\
\Leftrightarrow & \quad \{\ \text{images}\ \} \\
& c \in \mathsf{img}\, f
\end{array}$$

Conversely, assume that $\mathsf{img}\, f \supseteq \mathsf{img}\, h$ and $A \to B \neq \emptyset$, so that either $A = \emptyset$ or $B \neq \emptyset$. When $A = \emptyset$, then h is the empty function; let g be the empty function too, so $f \cdot g$ is also empty and hence equal to h. When $B \neq \emptyset$, we define $g\,a$ for $a \in A$ as follows. Let $c = h\,a$; by assumption, $c \in \mathsf{img}\, f$ too, so there exists $b \in B$ with $f\,b = c$, and we define $g\,a$ to be such a b. If there is more than one such b, it doesn't matter which one that we choose. By construction, this gives $h\,a = f\,(g\,a)$ for every a. $\square$

We also use the dual of Lemma 4.4:

Lemma 5.4

$$A \to \nu F \neq \emptyset \quad \Rightarrow \quad A \to F\,A \neq \emptyset$$

Proof. We note that $A \to F\,A \neq \emptyset$ is equivalent to $A \neq \emptyset \Rightarrow F\,A \neq \emptyset$, which implication can then be verified by combining the two calculations:

$$\begin{array}{ll} & A \neq \emptyset \\ \Rightarrow & \quad \{\ A \to \nu F \neq \emptyset\ \} \\ & \nu F \neq \emptyset \\ \Rightarrow & \quad \{\ \mathsf{out} : \nu F \to F\,(\nu F)\ \} \\ & F\,(\nu F) \neq \emptyset \end{array}$$

and

$$\begin{array}{ll} & A \neq \emptyset \\ \Rightarrow & \quad \{\ \text{functions}\ \} \\ & \nu F \to A \neq \emptyset \\ \Rightarrow & \quad \{\ \text{functors}\ \} \\ & F\,(\nu F) \to F\,A \neq \emptyset \end{array}$$

That is, $A \neq \emptyset$ implies that $F\,(\nu F) \neq \emptyset$ and $F\,(\nu F) \to F\,A \neq \emptyset$, which conjunction in turn implies that $F\,A \neq \emptyset$, as required. □

Proof of Theorem 5.2 Again, the proof is simple:

$$\begin{array}{ll} & \exists g : A \to F\,A.\quad h = \mathsf{unfold}\,g \\ \Leftrightarrow & \quad \{\ \text{universal property}\ \} \\ & \exists g : A \to F\,A.\quad \mathsf{out} \cdot h = F\,h \cdot g \\ \Leftrightarrow & \quad \{\ \text{Lemma 5.3}\ \} \\ & \mathsf{img}\,(F\,h) \supseteq \mathsf{img}\,(\mathsf{out} \cdot h)\ \wedge\ A \to F\,A \neq \emptyset \\ \Leftrightarrow & \quad \{\ \text{Lemma 5.4},\ h : A \to \nu F\ \} \\ & \mathsf{img}\,(F\,h) \supseteq \mathsf{img}\,(\mathsf{out} \cdot h) \end{array}$$

□

Remark 5.5 For the type $\mathit{Stream}(A)$ of streams with elements drawn from A, with destructors $\mathit{head} : \mathit{Stream}(A) \to A$ and $\mathit{tail} : \mathit{Stream}(A) \to \mathit{Stream}(A)$, Theorem 5.2 reduces to stating that an arbitrary function $h : B \to \mathit{Stream}(A)$ can be written directly as an unfold precisely when the *tail* of every stream producible by h is itself producible by h, in the sense that: $\mathsf{img}\,(\mathit{tail} \cdot h)\ \subseteq\ \mathsf{img}\,h$.

Example 5.6 Consider the function $\mathit{from} : \mathbb{N} \to \mathit{Stream}(\mathbb{N})$ defined in Section 2.2. Then $(\mathit{tail} \cdot \mathit{from})\, n$ is the stream $[n+1, n+2, \ldots]$, and in general, $\mathsf{img}\,(\mathit{tail} \cdot \mathit{from})$ is the set of streams $\{\ [n+1, n+2, \ldots] \mid n \in \mathbb{N}\ \}$, which is in included in $\mathsf{img}\,\mathit{from}$, the set of streams $\{\ [n, n+1, \ldots] \mid n \in \mathbb{N}\ \}$. Hence, *from* can be written directly using unfold.

Example 5.7 In contrast, if we define a function $\mathit{mults} : \mathbb{N} \to \mathit{Stream}(\mathbb{N})$ such that $\mathit{mults}\, n$ produces the stream of multiples $[0, n, n \times 2, n \times 3, \ldots]$ of a natural n, then $(\mathit{tail} \cdot \mathit{mults})\, n$ is the stream $[n, n{\times}2, \ldots]$, and so $\mathsf{img}\,(\mathit{tail} \cdot \mathit{mults})$ is not included in $\mathsf{img}\,\mathit{mults}$, which only includes streams whose *head* is 0. Therefore *mults* cannot be written directly as an unfold.

Remark 5.8 For the type $\mathit{CoTree}(A)$ of infinite binary trees with elements drawn from A, with destructors $\mathit{root} : \mathit{CoTree}(A) \to A$ and $\mathit{left}, \mathit{right} : \mathit{CoTree}(A) \to \mathit{CoTree}(A)$, Theorem 5.2 reduces to stating that an arbitrary function $h : B \to \mathit{CoTree}(A)$ can be written as an unfold precisely when the *left* and *right* of every tree producible by h are themselves producible by h:

$$\mathsf{img}\,(\mathit{left} \cdot h) \ \subseteq\ \mathsf{img}\, h$$

$$\mathsf{img}\,(\mathit{right} \cdot h) \ \subseteq\ \mathsf{img}\, h$$

Example 5.9 Consider the infinite binary tree with every node labelled by its *path*, a finite list of booleans recording the left and right turns from the root in order to reach that node. The function $\mathit{paths} : \mathbf{1} \to \mathit{CoTree}(\mathit{List}(\mathbb{B}))$ that produces this tree is not an unfold, because $\mathsf{img}\,(\mathit{left} \cdot \mathit{paths})$ and $\mathsf{img}\,(\mathit{right} \cdot \mathit{paths})$ contain trees with singleton lists at their roots, which are not included in $\mathsf{img}\,\mathit{paths}$, which contains a tree with the empty list at its root.

Example 5.10 In contrast, the more general function $\mathit{pathsfrom} : \mathit{List}(\mathbb{B}) \to \mathit{CoTree}(\mathit{List}(\mathbb{B}))$ that generates the tree of paths starting from a given path is an unfold, because $(\mathit{left} \cdot \mathit{pathsfrom})\, \mathit{bs} = \mathit{pathsfrom}\,(\mathit{false} : \mathit{bs})$ implies that $\mathsf{img}\,(\mathit{left} \cdot \mathit{pathsfrom})$ is included in $\mathsf{img}\,\mathit{pathsfrom}$, and similarly for *right*.

6 Conclusion

We have given the first complete results for when an arbitrary arrow can be written directly as a fold or unfold, for the special case of the category $\mathcal{SET}$. In future work we will investigate whether the results can be generalised to other categories, and to other patterns of recursion, such as primitive (co-)recursion [19,22] and course-of-value (co-)iteration [23].

As well as being interesting from a theoretical point of view, we also expect the results to have practical applications in program optimisation. A well-structured program is typically factored into several phases, each phase generating a data structure that is consumed by the subsequent phase; *deforestation* [9,16,21] fuses adjacent phases and eliminates the intermediate data structures. When performed as a compiler optimisation, it yields efficient object code without sacrificing the structure and clarity of the source code. Our results can be used to determine when two phases cannot be fused to a fold or an unfold. It might be possible to use an automatic testing system such as QuickCheck [2] to find counterexamples to the appropriate inclusions.

Acknowledgements

We are very grateful to Lambert Meertens, whose suggestions lead to a substantial simplification of our proofs. We also thank the anonymous referees for their useful comments. Graham Hutton was supported by EPSRC grant *Structured Recursive Programming*, and together with Thorsten Altenkirch by ESPRIT Working Group *Applied Semantics*.

References

[1] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[2] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, September 2000.

[3] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[4] J. Gibbons. Calculating functional programs. In *Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Oxford, April 2000.

[5] J. Gibbons and G. Hutton. Proof methods for structured corecursive programs. In *Proc. 1st Scottish Functional Programming Workshop*, Stirling, Scotland, August 1999.

[6] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[7] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, 1992.

[8] S. Gorlatch. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming*, 33:1–27, 1999.

[9] Z. Hu, H. Iwasaki and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proc. 1st ACM SIGPLAN International Conference on Functional Programming*, 1996.

[10] G. Hutton. Fold and unfold for program semantics. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[11] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[12] B. Jacobs. Exercises in coalgebraic specification. In *Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Oxford, April 2000.

[13] B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors. *Proc. of the First Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1998. Electronic Notes in Theoretical Computer Science Volume 11.

[14] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[15] B. Jacobs and J. Rutten, editors. *Proc. of the Second Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1999. Electronic Notes in Theoretical Computer Science Volume 19.

[16] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1995.

[17] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[18] G. Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14(2-3):255–280, September 1990.

[19] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

[21] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1995.

[22] V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998.

[23] V. Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis, Universitity of Tartu, 2000.

[24] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.