Under consideration for publication in Formal Aspects of Computing

Disciplined, efficient, generalised folds for nested datatypes

Clare Martin[1], Jeremy Gibbons[2] and Ian Bayley[2]

[1]School of Computing and Mathematical Sciences, Oxford Brookes University.
[2]Oxford University Computing Laboratory.

Abstract. Nested (or non-uniform, or non-regular) datatypes have recursive definitions in which the type parameter changes. Their folds are restricted in power due to type constraints. Bird and Paterson introduced *generalised folds* for extra power, but at the cost of a loss of efficiency: folds may take more than linear time to evaluate. Hinze introduced *efficient generalised folds* to counter this inefficiency, but did so in a pragmatic way: he did not provide categorical or equivalent underpinnings, so did not get the associated universal properties for manipulating folds. We combine the efficiency of Hinze's construction with the powerful reasoning tools of Bird and Paterson's.

Keywords: Nested datatype, non-uniform datatype, non-regular datatype, polymorphic recursion, fold, universal property, functor category.

1. Introduction

The fold operator of functional programming can be defined on any regular datatype [BdM97], but there is a problem when extending its definition to nested datatypes [BM98] because some of its parameters are required to be polymorphic. A solution to this problem, proposed by Bird and Paterson [BP99b], was to introduce a new kind of operator called a *generalised fold*. Like an ordinary fold, the generalised fold of a given initial algebra is characterised uniquely by its defining equation, and this characterisation gives rise to useful fusion laws. Unfortunately generalised folds still lack some of the other well-known properties of folds: they cannot be used to define map functions, and the map fusion law is subject to a side condition. A more serious shortcoming, observed by Hinze [Hin00], is that generalised folds are inefficient for some datatypes. So Hinze has introduced a different kind of fold, called an *efficient generalised fold*, which suffers none of the problems described above. It is specified as a generalised fold composed with a map, but the generalised fold is different from that defined in [BP99b]. (The difference will be explained in Section 6.)

This paper is written in response to Hinze's remark [Hin00] that he did not know whether the generalised folds of [BP99b] also have an efficient counterpart. We show that they do, by giving a unique characterisation of efficient folds in terms of initial algebras. This universal property provides a proof principle that can be used to show that the efficient fold can be expressed as a Bird/Paterson generalised fold after a map, and consequently

Correspondence and offprint requests to: Clare Martin, School of Computing and Mathematical Sciences, Wheatley Campus, Oxford Brookes University, Oxford OX33 1HX, UK. Email: `cemartin@brookes.ac.uk`.

we obtain the fusion laws for free. It is not clear whether there is a similar universal construction of Hinze's folds.

We begin, in Section 2, by recalling some of the well-known properties of folds on regular datatypes, exemplified by the familiar datatype of lists. In Section 3, we discuss the relative merits of ordinary folds, generalised folds and efficient folds on the nested datatype of perfectly balanced binary trees. The time complexity of the latter two folds is then compared on a simple random access list [Oka98] in Section 4. Sections 5, 6 and 7 contain a short description of the semantics of nested datatypes, generalised folds and efficient folds respectively. Some applications of efficient folds are given in Section 8.

2. Fold on Regular Datatypes

We will use the datatype of lists to exhibit some of the intrinsic properties of folds on regular datatypes. The most fundamental of these is the universal property of fold, from which the fold fusion law for program transformation can be derived. In addition, the map function of any regular datatype can be expressed as a fold. As such, it can be fused with any other fold, yielding a second law for program derivation, usually known as the map fusion law.

The datatype of lists can be defined in the programming language Haskell [PJ03] by

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ (a, \mathit{List}\ a)$$

and its fold operation can be defined recursively by

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{fold} :: \forall a.\forall b.\ b \to ((a,b) \to b) \to \mathit{List}\ a \to b} \\ \mathit{fold}\ e\ f\ \mathit{Nil} & = & e \\ \mathit{fold}\ e\ f\ (\mathit{Cons}\ (x, \mathit{xs})) & = & f\ (x, \mathit{fold}\ e\ f\ \mathit{xs}) \end{array}$$

(For clarity, we will be explicit about all universal quantifications over types; Haskell 98 allows implicit outermost universal quantifications.) The universal property of fold states that, for finite lists, the function *fold e f* is the unique solution of this pair of equations: if $u : \mathit{List}\ a \to b$, $e : b$ and $f : (a,b) \to b$,

$$\begin{array}{ll} & u = \mathit{fold}\ e\ f \\ \Leftrightarrow & \\ & u\ \mathit{Nil} = e \text{ and } u\ (\mathit{Cons}\ (x, \mathit{xs})) = f\ (x, u\ \mathit{xs}) \text{ for all } x, \mathit{xs} \end{array} \tag{1}$$

This property is a useful proof principle for program transformation since it captures the familiar pattern of inductive proof for finite lists. In particular, it can be used to derive the following fusion law for lists, which gives conditions under which the composition of an arbitrary function and a fold can be fused into a single fold: if $k : b \to b'$, $e : b$, $f : (a,b) \to b$, $e' : b'$ and $f' : (a,b') \to b'$, then

$$\begin{array}{ll} & k\ e = e' \text{ and } k \cdot f = f' \cdot \mathit{id} \times k \\ \Rightarrow & \\ & k \cdot \mathit{fold}\ e\ f = \mathit{fold}\ e'\ f' \end{array} \tag{2}$$

The map fusion law for lists can be derived from this law, together with the definition of the map function as a fold. We will refer to the map function on lists as *list* for consistency with later examples. It can be expressed as a fold in the following way:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{list} :: \forall a.\forall b.\ (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b} \\ \mathit{list}\ f & = & \mathit{fold}\ \mathit{Nil}\ (\mathit{Cons} \cdot (f \times \mathit{id})) \end{array}$$

This definition might not be as readable as one written using explicit recursion, but it does have the advantage that properties of *list* can be proved from more fundamental ones of *fold*. The map fusion law below is an example of this. It would normally have been proved by induction, but is now almost immediate from the fold fusion law (2): if $e : b$, and $f : (a,b) \to b$, then for all $k : a' \to a$,

$$\mathit{fold}\ e\ f \cdot \mathit{list}\ k \quad = \quad \mathit{fold}\ e\ (f \cdot (k \times \mathit{id})) \tag{3}$$

This law does not hold for generalised folds, but it does hold for efficient folds, as we shall see in Sections 3 and 7.

3. Folds on Nested Datatypes

Nested datatypes are recursively defined parameterised datatypes in which the parameter of the datatype changes in the recursive call. For example, the *Nest* datatype, which is a level-wise representation of node-oriented, perfectly balanced binary trees, can be defined in Haskell by

$$\begin{array}{lcl} \textbf{data}\ \mathit{Nest}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ (a, \mathit{Nest}\ (\mathit{Pair}\ a)) \\ \textbf{type}\ \mathit{Pair}\ a & = & (a,a) \end{array} \tag{4}$$

This is a nested datatype because the *Nest* constructor is given the parameter *a* on the left hand side of this equation and a different argument, *Pair a*, on the right. The example below shows how this datatype represents perfectly balanced trees:

$$\begin{array}{lcl} \mathit{example} & :: & \mathit{Nest\ Int} \\ \mathit{example} & = & \mathit{Cons}\ (1,\ \mathit{Cons}\ ((2,3),\ \mathit{Cons}\ (((4,5),(6,7)),\ \mathit{Nil}))) \end{array} \tag{5}$$

3.1. Ordinary Folds

One standard function that we might wish to apply to a nest of integers is the *sum* function, to sum the values stored in it. The computation of the *sum* function is an instance of a more general pattern called a *reduction*. Reductions can be defined on any regular datatype using folds alone, but this is not true of nested datatypes. The initial algebra semantics of nested datatypes proposed in [BM98, MG01] gives the following definition of *fold* on this datatype:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{fold} :: \forall n.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (a, n\ (\mathit{Pair}\ a)) \to n\ a) \to \mathit{Nest}\ b \to n\ b} \\ \mathit{fold}\ e\ f\ \mathit{Nil} & = & e \\ \mathit{fold}\ e\ f\ (\mathit{Cons}\ (x,xs)) & = & f\ (x, \mathit{fold}\ e\ f\ xs) \end{array}$$

where *b* and *n* range over types and type constructors (functors) respectively. The universal property associated with this definition is identical to that for lists (1) apart from the type declarations. The problem with this definition is that it is limited in the scope of its applicability, because the second parameter must be polymorphic. So it can be used to implement some polymorphic reductions, like that of flattening a nest to a list, but not usually for monomorphic reductions like *sum*. We can illustrate the difficulty in expressing *sum* as a fold on the above example (5). The first parameter, e, must be zero, since it represents the sum of the empty nest. Writing the second parameter f as an infix operator $\oplus$ and using the Haskell syntax of writing the prefix form of an infix operator $\oplus$ in parentheses $(\oplus)$, we have

$$\mathit{fold}\ 0\ (\mathit{uncurry}\ (\oplus))\ \mathit{example} \quad = \quad 1 \oplus ((2,3) \oplus (((4,5),(6,7)) \oplus 0))$$

The function $(\oplus)$ must have type $\forall a.a \to \mathit{Int} \to \mathit{Int}$ for this fold to return an integer value, because it must be applied to values of a different type at each recursive call. The naturality property associated with values of this type in Haskell means that they must ignore the first parameter; therefore *sum* cannot be written as a *fold*.

3.2. Generalised Folds

Generalised folds provide a solution to the type problem encountered with ordinary folds, but their definitions depend on those of the corresponding map functions. The map function for *Nest* is:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{nest} :: \forall a.\forall b.\ (a \to b) \to \mathit{Nest}\ a \to \mathit{Nest}\ b} \\ \mathit{nest}\ f\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{nest}\ f\ (\mathit{Cons}\ (x,xs)) & = & \mathit{Cons}\ (f\ x, \mathit{nest}\ (\mathit{pair}\ f)\ xs) \end{array}$$

where the map for *Pair* is simply

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{pair} :: \forall a.\forall b.\ (a \to b) \to \mathit{Pair}\ a \to \mathit{Pair}\ b} \\ \mathit{pair}\ f\ (x,y) & = & (f\ x, f\ y) \end{array}$$

The generalised fold on nests is then defined below. It has one more parameter than the ordinary fold, and this extra parameter is repeatedly mapped over the datatype at each recursive call; this is the cause of the inefficiency observed in [Hin00] and reiterated here in Section 4.

$$\begin{array}{lcl} \mathit{gfold} :: \forall n.\forall m.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (m\ a, n\ (\mathit{Pair}\ a)) \to n\ a) \to (\forall a.\ \mathit{Pair}\ (m\ a) \to m\ (\mathit{Pair}\ a)) \\ \qquad \to \mathit{Nest}\ (m\ b) \to n\ b \\ \mathit{gfold}\ e\ f\ g\ \mathit{Nil} & = & e \\ \mathit{gfold}\ e\ f\ g\ (\mathit{Cons}\ (x, xs)) & = & f\ (x, \mathit{gfold}\ e\ f\ g\ (\mathit{nest}\ g\ xs)) \end{array} \tag{6}$$

This definition is essentially the same as that in [Hin00], except that Hinze gives a more general type to the second and third parameters:

$$\begin{array}{l} \mathit{gfold} :: \forall n.\forall m.\forall p.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (m\ a, n\ (p\ a)) \to n\ a) \to (\forall a.\ \mathit{Pair}\ (m\ a) \to m\ (p\ a)) \\ \qquad \to \mathit{Nest}\ (m\ b) \to n\ b \end{array}$$

More substantial differences can be seen on other datatypes, as we will show in Section 6.2.

An ordinary fold is a special case of a generalised fold, obtained by setting function parameter g to the identity function and type parameter m to the identity functor. The difference between the two folds is most clearly shown by applying a generalised fold to the example nest of integers again. Writing the parameter g as an infix operator $\otimes$, and using the Haskell function *uncurry* : $(a \to (b \to c)) \to ((a, b) \to c)$ we have

$$\mathit{gfold}\ e\ (\mathit{uncurry}\ (\oplus))\ (\mathit{uncurry}\ (\otimes))\ \mathit{example} \quad = \quad 1 \oplus ((2 \otimes 3) \oplus (((4 \otimes 5) \otimes (6 \otimes 7)) \oplus e)) \tag{7}$$

It is now evident that any reduction on nests can be written as a generalised fold, by taking $\otimes$ to be $\oplus$. In particular, the *sum* function can be written as:

$$\mathit{sum} = \mathit{gfold}\ 0\ (\mathit{uncurry}\ (+))\ (\mathit{uncurry}\ (+))$$

This function cannot be implemented in Haskell with the most general type signature for the generalised fold: the type must be instantiated to different values for different applications. In this case we must use the type

$$\mathit{gfold} :: \forall a.\forall b.\ b \to ((a, b) \to b) \to (\mathit{Pair}\ a \to a) \to \mathit{Nest}\ a \to b$$

This problem is due to weaknesses in Haskell's type checker, which implements decidable type inference by using a kinded first-order unification rather than full higher-order unification [Jon95], which is only semi-decidable [Hue75]. This is one complication that is not removed by using efficient folds.

3.3. Fusion

We have now shown how two of the three problems with generalised folds that were listed in the introduction manifest themselves on the datatype of nests: the generalised fold is defined in terms of the map function, rather than vice versa, and the generalised fold can be inefficient because of the use of the map function in the recursive call. The third problem concerned the map fusion law, which we will consider in this section: the law is subject to a side condition. Before doing so, we will observe that the fold fusion law for generalised folds can be derived from the universal property, just as it can for ordinary folds on regular datatypes.

Fold Fusion

The universal property of generalised folds on nests states that for all functors m and n, and all $u : \forall a.\ \mathit{Nest}\ (m\ a) \to (n\ a)$, $e : \forall a.\ n\ a$, $f : \forall a.\ (m\ a, n\ (\mathit{Pair}\ a)) \to n\ a$ and $g : \forall a.\ \mathit{Pair}\ (m\ a) \to m\ (\mathit{Pair}\ a)$,

$$\begin{array}{ll} & u = \mathit{gfold}\ e\ f\ g \\ \Leftrightarrow & \\ & u\ \mathit{Nil} = e \text{ and } u\ (\mathit{Cons}\ (x, xs)) = f\ (x, u\ (\mathit{nest}\ g\ xs)) \text{ for all } x, xs \end{array} \tag{8}$$

A fusion law identical to that for lists (2) can be derived from this universal property alone, but a stronger law is also derivable by observing that the naturality of u implies that for functors m and n and for all $k' : a \to b$,

$$u \cdot \mathit{nest}\ (m\, k') \quad = \quad (n\, k') \cdot u \tag{9}$$

Combining (9) with the universal property (8) gives the following law: if $e' : \forall a.\ n'\ a, f' : \forall a.\ (m\ (m'\ a), n'\ (m'\ (Pair\ a))) \to n'\ (m'\ a), k : \forall a.\ n\ a \to n'\ a$, and $k' : \forall a.\ Pair\ (m'\ a) \to m'\ (Pair\ a)$ then

$$\begin{array}{ll} & k\ e = e' \text{ and } k \cdot f = f' \cdot (id \times (k \cdot (n \cdot k'))) \\ \Rightarrow & \\ & k \cdot gfold\ e\ f\ g = gfold\ e'\ f'\ (m\ k' \cdot g) \end{array} \tag{10}$$

The simpler law corresponding to (2) is recaptured by setting k' to the identity function and m' to the identity functor.

Map Fusion

Recall that the map fusion law for lists (3) was derived from the fold fusion law together with the expression of the map function as a fold. Clearly this is not possible for generalised folds, because their definition depends on that of the map function; indeed it is not always the case that a generalised fold composed with a map can be rewritten as another generalised fold, as the following example illustrates:

$$sumsquares \quad = \quad gfold\ 0\ (uncurry\ (+))\ (uncurry\ (+)) \cdot nest\ (\lambda x.\ x * x)$$

There are cases where fusion is still possible though, such as the *revels* function below which reverses the order of the labels in each level of a tree of pairs:

$$revels \quad = \quad gfold\ Nil\ Cons\ swap \cdot nest\ swap$$

where $swap\ (x, y) = (y, x)$. This can be fused using the map fusion law for nests of [BP99b] which is easily derivable from (8) and gives the following sufficient condition for fusion: if $k :: \forall a.\ m'\ a \to m\ a$ and $g' :: \forall a.\ Pair\ (m'\ a) \to m'\ (Pair\ a)$, then

$$\begin{array}{ll} & g \cdot pair\ k = k \cdot g' \\ \Rightarrow & \\ & gfold\ e\ f\ g \cdot nest\ k = gfold\ e\ (f \cdot (k \times id))\ g' \end{array} \tag{11}$$

This condition is certainly not necessary for fusing a map with a generalised fold. For example, it applies to neither of the following functions, yet both can be expressed as a single generalised fold:

$$\begin{array}{lcl} flatten & = & gfold\ [\,]\ (uncurry\ (+\!+))\ (uncurry\ (+\!+)) \cdot nest\ (\lambda x.\ [x]) \\ size & = & gfold\ 0\ (uncurry\ (+))\ (uncurry\ (+)) \cdot nest\ (\lambda x.\ 1) \end{array}$$

where $+\!+$ denotes list concatenation. We will see in Section 8 that fusion laws (10) and (11) can be combined to give a single *fold equivalence* law that does apply to examples like *flatten* and *size*, but we have yet to derive a generic version of the law.

3.4. Efficient Folds

Now we will introduce the efficient fold on nests, and see that it is not necessary to put any conditions on its map fusion law. The definition is as follows:

$$\begin{array}{lcl} efold :: \forall n.\forall m.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (m\ a, n\ (Pair\ a)) \to n\ a) \to (\forall a.\ Pair\ (m\ a) \to m\ (Pair\ a)) \\ \qquad\qquad \to (\forall l.\forall z.(l\ b \to m\ (z\ b)) \to Nest\ (l\ b) \to n\ (z\ b)) \\ efold\ e\ f\ g\ h\ Nil & = & e \\ efold\ e\ f\ g\ h\ (Cons\ (x, xs)) & = & f\ (h\ x, efold\ e\ f\ g\ (g \cdot pair\ h)\ xs) \end{array} \tag{12}$$

Notice that the types of the first three parameters are the same as in the corresponding generalised fold. Like the generalised fold on nests, this definition differs from that of [Hin00] only in its type signature, which Hinze declared as:

$$\begin{array}{l} efold :: \forall n.\forall m.\forall p.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (m\ a, n\ (p\ a)) \to n\ a) \to (\forall a.\ Pair\ (m\ a) \to m\ (p\ a)) \\ \qquad\qquad \to (\forall l.\forall z.(l\ (m\ b) \to m\ (z\ b)) \to Nest\ (l\ (m\ b)) \to n\ (z\ b)) \end{array}$$

Hinze pointed out that, at the time of writing his paper, efficient folds were not yet implementable in Haskell, but the latest version of the Glasgow Haskell Compiler does accept definition (12) and its counterparts for other nested datatypes.

The efficient fold has a shorter definition than the corresponding generalised fold, since it does not require the definition of the *nest* function. Indeed, *nest* can be implemented as an efficient fold: for all $h : l\ a \to m\ b$,

$$\mathit{nest}\ h \quad = \quad \mathit{efold}\ \mathit{Nil}\ \mathit{Cons}\ \mathit{id}\ h \tag{13}$$

Clearly the identity function is a special case of this. This definition of *nest* is not surprising when we consider the action of the efficient fold on the previous example of a nest of integers:

$$\mathit{efold}\ e\ (\mathit{uncurry}\ (\oplus))\ (\mathit{uncurry}\ (\otimes))\ h\ \mathit{example} = h\ 1 \oplus ((h\ 2 \otimes h\ 3) \oplus (((h\ 4 \otimes h\ 5) \otimes (h\ 6 \otimes h\ 7)) \oplus e))$$

Comparing this with (7) suggests the following more general property of the efficient fold on nests: for all e, f, g and h of appropriate type,

$$\mathit{efold}\ e\ f\ g\ h \quad = \quad \mathit{gfold}\ e\ f\ g \cdot \mathit{nest}\ h \tag{14}$$

A property similar to this was given as the specification of efficient folds in [Hin00]. Instead, we observe that both (13) and (14) can be derived from the following universal property of efficient folds: for all type constructors m and n, and all $u : \forall a.\forall l.\forall z.\ (l\ a \to m\ (z\ a)) \to \mathit{Nest}\ (l\ a) \to (n\ (z\ a))$, $e : \forall a.\ n\ a$, $f : \forall a.\ m\ a \to n\ (\mathit{Pair}\ a) \to n\ a$ and $g : \forall a.\ \mathit{Pair}\ (m\ a) \to m\ (\mathit{Pair}\ a)$,

$$\begin{array}{ll} & u = \mathit{efold}\ e\ f\ g \\ \Leftrightarrow & \\ & u\ h\ \mathit{Nil} = e \text{ and } u\ h\ (\mathit{Cons}\ (x, xs)) = f\ (h\ x, u\ (g \cdot \mathit{pair}\ h)\ xs) \text{ for all } x, xs, \text{ and all } h : l\ a \to m\ (z\ a). \end{array}$$

This property is extended to an initial algebra definition of efficient folds on arbitrary nested datatypes in Section 7, where it is proved that they can always be written as a generalised fold after a map. It follows that any function that could be written as a generalised fold could also be written as an efficient fold. Moreover, it is now clear from (14), together with the functorial property of *nest*, that in addition to fusion laws corresponding to (10) and (11), efficient folds satisfy the following simple map fusion law: for all $k :: \forall a.\ l'\ a \to l\ a$,

$$\mathit{efold}\ e\ f\ g\ h \cdot \mathit{nest}\ k \quad = \quad \mathit{efold}\ e\ f\ g\ (h \cdot k) \tag{15}$$

This law is similar to the map fusion law for ordinary folds, which suggests that the pattern of computation captured by ordinary folds may be more closely mimicked by efficient folds than by generalised folds. Notice however that the derivation of law (15) is quite different from its regular counterpart; in fact, neither (11) nor (15) can be deduced solely from the universal property of efficient folds.

4. Efficiency

Efficient folds were described on the datatype of nests in Section 3 because it is one of the most simple non-regular datatypes that can be defined. Unfortunately it is not a suitable datatype with which to demonstrate how performance can be increased by using efficient folds, because they are not asymptotically faster than generalised folds on nests. It was observed in [Hin00] that one datatype for which efficient folds do give a worthwhile improvement is that of de Bruijn terms [BP99a]. In this section we will describe how a similar benefit is gained on a datatype that is more closely related to nests: random access lists. Random access lists were invented by Okasaki [Oka98] as a means of improving the efficiency of certain operations on traditional lists.

The Haskell definition of a random access list given below contains the type constructors *Nil* and *One* corresponding to *Nil* and *Cons* in the definition of *Nest* (4), together with an additional type constructor *Zero*.

$$\textbf{data}\ \mathit{Ral}\ a \quad = \quad \mathit{Nil} \mid \mathit{Zero}\ (\mathit{Ral}\ (\mathit{Pair}\ a)) \mid \mathit{One}\ (a, \mathit{Ral}\ (\mathit{Pair}\ a))$$

This datatype can represent collections of any size, unlike the *Nest* datatype which is restricted to representing collections of size $2^k - 1$ for some k. A collection of data of size n is represented by a sequence of tuples of sizes 2^i for each i corresponding to a one in the binary representation of n. For example, the sequence $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ has length 11, so it could be stored as:

$$\mathit{One}\ (0, \mathit{One}\ ((1,2), \mathit{Zero}\ (\mathit{One}\ ((((3,4),(5,6)),((7,8),(9,10))), \mathit{Nil}))))$$

Notice how the constructors $\mathit{One}, \mathit{One}, \mathit{Zero}, \mathit{One}$ spell out the binary expansion of 11, least significant bit first. Like binary numbers, random access lists can contain any number of trailing zeros. So, for example, in any such data structure, the *Nil* may be replaced by $\mathit{Zero}^n\ \mathit{Nil}$ for any n without changing the collection represented, and in particular, the empty collection of values could be represented by $Zero^n\ Nil$ for any n. We will use this empty collection to compare the relative efficiency of the generalised and efficient folds on this datatype. In order to do so, we must first give their definitions, together with that of the map function. The types have been omitted from the definitions below for conciseness, since it is the pattern of computation, rather than the type, that is relevant to efficiency.

$$\begin{array}{lcl}
ral\ f\ Nil & = & Nil \\
ral\ f\ (Zero\ xs) & = & Zero\ (ral\ (pair\ f)\ xs) \\
ral\ f\ (One\ (x,xs)) & = & One\ (f\ x, ral\ (pair\ f)\ xs) \\
\\
gfold\ e\ f\ g\ h\ Nil & = & e \\
gfold\ e\ f\ g\ h\ (Zero\ xs) & = & f\ (gfold\ e\ f\ g\ h\ (ral\ h\ xs)) \\
gfold\ e\ f\ g\ h\ (One\ (x,xs)) & = & g\ (x, gfold\ e\ f\ g\ h\ (ral\ h\ xs)) \\
\\
efold\ e\ f\ g\ h\ k\ Nil & = & e \\
efold\ e\ f\ g\ h\ k\ (Zero\ xs) & = & f\ (efold\ e\ f\ g\ h\ (h \cdot pair\ k)\ xs) \\
efold\ e\ f\ g\ h\ k\ (One\ (x,xs)) & = & g\ (k\ x, efold\ e\ f\ g\ h\ (h \cdot pair\ k)\ xs)
\end{array}$$

At first sight, these definitions may seem rather daunting, because they have such a large number of parameters. In this case, however, the following specialisations (in which $f = id$, so *Zero* constructors do not contribute towards the result, and $h = g$, so the pairs in the 'tail' are collapsed using the same binary operator as combines the head with the result of the recursive call) are often useful:

$$\begin{array}{lcl}
greduce\ e\ g & = & gfold\ e\ id\ g\ g \\
ereduce\ e\ g\ k & = & efold\ e\ id\ g\ g\ k
\end{array}$$

The *sum* function can be implemented by either of these, taking e, g and k to be 0, *plus* and *id* respectively, where $plus = uncurry\ (+)$, but the first program takes quadratic time whereas the second is linear. This can be seen by first considering the evaluation of $greduce\ 0\ plus\ (Zero^n\ Nil)$. At the ith step, the expression

$$(greduce\ 0\ plus) \cdot ral\ plus \cdot ral\ (pair\ plus) \cdot \ldots \cdot ral\ (pair^{i-1}\ plus)\ (Zero^{n-i}\ Nil)$$

must be simplified to

$$(greduce\ 0\ plus) \cdot Zero \cdot ral\ (pair\ plus) \cdot ral\ (pair^2\ plus) \cdot \ldots \cdot ral\ (pair^i\ plus)\ (Zero^{n-i-1}\ Nil)$$

in order for computation to proceed. Since this takes $O(i)$ time, the total running time is $O(n^2)$. Now consider the evaluation of $ereduce\ 0\ plus\ id\ (Zero^n\ Nil)$. At the ith step, the expression

$$ereduce\ 0\ plus\ (plus \cdot pair\ (plus \cdot (pair\ (\ldots))))\ (Zero^{n-i}\ Nil)$$

must be simplified to

$$ereduce\ 0\ plus\ (plus \cdot pair\ (plus \cdot pair\ (plus \cdot (pair\ (\ldots)))))\ (Zero^{n-i-1}\ Nil)$$

in order for computation to proceed. Since this takes constant time, the total running time is $O(n)$.

It might be argued that this example is unconvincing, relying as it does on unnormalized random-access lists (those with 'trailing zeros'). On normalized random-access lists, the speedup is only from $O(n \log n)$ to $O(n)$, which is less dramatic but nevertheless an asymptotic speedup.

5. Semantics of Nested Datatypes

Having motivated the introduction of efficient generalised folds, we turn now to their formalisation. We base our presentation on the semantics for nested datatypes presented in [MG01], which is restricted to one argument datatypes with no mutual recursion. (The generalisation to multiple arguments and mutual recursion is straightforward.)

Suppose that **C** is a ω-cocomplete category with initial object and all finite products and coproducts, where $\omega = \{0 \to 1 \to 2 \to 3 \to \ldots\}$. Then the class of nested datatypes considered here is the closure under initial algebras of the class of higher-order functors or *hofunctors* defined below. Note that this class does not include all Haskell datatypes, since the operators listed below are just the standard ones for constructing polynomial datatypes [MA86] together with a horizontal composition operator. So, it does not include rose trees [Mee87] for example. Let **Coc(C)** denote the category with objects all ω-cocontinuous endofunctors on **C** and arrows all natural transformations between them. The horizontal composition bifunctor $\star : \mathbf{Coc}(\mathbf{C}) \times \mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ maps each pair of functors to its ordinary composite, and each pair of natural transformations $\theta :: P \to Q$ and $\psi :: R \to S$ to the horizontal composite $\theta \star \psi :: P \cdot R \to Q \cdot S$ defined by

$$\theta \star \psi = \theta S \cdot P\psi = Q\psi \cdot \theta R$$

We will call a *nested* hofunctor **Coc(C)**$\to$ **Coc(C)** any hofunctor which can be constructed from the identity hofunctor and constant hofunctors, through the use of the horizontal composition, product or coproduct operators, as listed below in decreasing order of precedence.

1. $\mathscr{I}d$, the identity functor;
2. $\mathscr{K}_Q$, the constant functor for each object Q;
3. $\mathscr{F} \dot\star \mathscr{G}$, the horizontal composition of $\mathscr{F}$ and $\mathscr{G}$;
4. $\mathscr{F} \ddot\times \mathscr{G}$, the product of $\mathscr{F}$ and $\mathscr{G}$;
5. $\mathscr{F} \ddot+ \mathscr{G}$, the coproduct of $\mathscr{F}$ and $\mathscr{G}$.

The last three operators are defined pointwise in terms of the corresponding bifunctors in $\mathbf{Coc}(\mathbf{C})$. Given any bifunctor $\oplus : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, the bifunctor $\dot\oplus : \mathbf{Coc}(\mathbf{C}) \times \mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ is defined for all objects F, G and arrows θ, ψ of $\mathbf{Coc}(\mathbf{C})$ by

$$\begin{array}{lcl} (F \dot\oplus G)\, c & = & F\, c \oplus G\, c \\ (F \dot\oplus G)\, f & = & F\, f \oplus G\, f \\ (\theta \dot\oplus \psi)\, c & = & \theta\, c \oplus \psi\, c \end{array}$$

for objects c and arrows f of $\mathbf{C}$. We use the notation $\ddot\oplus$ to denote operators that are lifted twice. In the sequel we will use the fact that clause 3 above can be replaced by $\mathscr{K}_Q \dot\star \mathscr{G}$ and $\mathscr{I}d \dot\star \mathscr{G}$, since for all $\mathscr{F}$, $\mathscr{G}$ and $\mathscr{H}$,

$$\begin{array}{lcl} (\mathscr{F} \ddot+ \mathscr{G}) \dot\star \mathscr{H} & = & (\mathscr{F} \dot\star \mathscr{H}) \ddot+ (\mathscr{G} \dot\star \mathscr{H}) \\ (\mathscr{F} \ddot\times \mathscr{G}) \dot\star \mathscr{H} & = & (\mathscr{F} \dot\star \mathscr{H}) \ddot\times (\mathscr{G} \dot\star \mathscr{H}) \\ (\mathscr{F} \cdot \mathscr{G}) \cdot \mathscr{H} & = & \mathscr{F} \cdot (\mathscr{G} \cdot \mathscr{H}) \end{array}$$

Note also that the grammar of hofunctors is not a free one as, for instance, $(\mathscr{F} \ddot+ \mathscr{G}) \cdot \mathscr{H} = (\mathscr{F} \cdot \mathscr{H}) \ddot+ (\mathscr{G} \cdot \mathscr{H})$.

As an example, we return to the *Nest* datatype, which can be defined as the initial algebra of the hofunctor $\mathscr{N} : \mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ given by

$$\mathscr{N} = \mathscr{K}_{K_1} \ddot+ (\mathscr{K}_{Id} \ddot\times (\mathscr{I}d \dot\star \mathscr{K}_{Pair})) \tag{16}$$

where the parentheses are redundant, thanks to the precedences chosen above. Since *Nest* is an initial algebra of $\mathscr{N}$, it satisfies $\mathscr{N}\ (Nest) \cong Nest$, because the initial algebra of any functor is a fixed point of that functor. This equation corresponds to the recursive definition of nests in Haskell (4).

6. Generalised Folds

The standard definition of the fold function (see eg [BdM97]) states that if $\mathbf{C}$ is a category and $F : \mathbf{C} \to \mathbf{C}$ is a functor with least fixed point T, then there exists an initial F-algebra $\alpha : FT \to T$ such that for all $v : FN \to N$ there exists a unique arrow $fold_F\, v$ where

$$fold_F\, v : T \to N$$

is characterised by the universal property that for all $u : T \to N$,

$$u = fold_F\, v \quad \Leftrightarrow \quad u \cdot \alpha = v \cdot F\, u$$

This definition was generalised in [BP99b] to the category **Coc(C)**. We will use different notation but the definition is essentially the same. For simplicity, we restrict attention to unary functors. We will denote an empty collection of parameters by $\langle\rangle$, the single parameter r by $\langle r \rangle$, and if $\overline{u}$ and $\overline{w}$ represent the collections $u_0, u_1, \ldots, u_m$ and $w_0, w_1, \ldots, w_n$ respectively, then $\overline{u} \frown \overline{w}$ is the collection $u_0, u_1, \ldots, u_m, w_0, w_1, \ldots, w_n$. Now let $\mathscr{F}$:

$\mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ be a nested hofunctor with least fixed point T and initial algebra $\alpha : \mathscr{F}T \to T$. The parameter collection $\overline{v}_{\mathscr{F}}$ of $\mathscr{F}$ is defined inductively by

$$\begin{array}{lcl}
\overline{v}_{\mathscr{K}_Q} & = & \langle\rangle \\
\overline{v}_{\mathscr{I}d} & = & \langle r\rangle \\
\overline{v}_{\mathscr{F}\ddot{+}\mathscr{G}} & = & \overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}} \\
\overline{v}_{\mathscr{F}\ddot{\times}\mathscr{G}} & = & \overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}} \\
\overline{v}_{\mathscr{K}_Q\dot{\star}\mathscr{G}} & = & \langle s\rangle \frown \overline{v}_{\mathscr{G}} \\
\overline{v}_{\mathscr{I}d\dot{\star}\mathscr{G}} & = & \langle t\rangle \frown \overline{v}_{\mathscr{G}}
\end{array}$$

for some parameters r, s and t, whose types are determined below. Note that different instances of each of the hofunctors $\mathscr{I}d$, $\mathscr{K}_Q \dot{\star} \mathscr{G}$ and $\mathscr{I}d \dot{\star} \mathscr{G}$ in $\mathscr{F}$ generate different instances of each of the parameters r, s and t. Now let $\overline{v} = \langle v\rangle \frown \overline{v}_{\mathscr{F}}$, for some further parameter v, then, given functors M and N, the generalised fold $gfold\,\overline{v}$ has type

$$gfold_{\mathscr{F}}\,\overline{v} : T \cdot M \to N$$

and is characterised uniquely, as proved in [BP99b], by the property that for all $u : T \cdot M \to N$

$$u = gfold_{\mathscr{F}}\,\overline{v} \quad \Leftrightarrow \quad u \cdot \alpha_M = v \cdot \Phi_{\mathscr{F}}\,\overline{v}_{\mathscr{F}}\; u \tag{17}$$

where $\Phi_{\mathscr{F}}\,\overline{v}_{\mathscr{F}} : (T \cdot M \to N) \to \mathscr{F}T \cdot M \to \mathscr{R}_{\mathscr{F}}(M,N)$, $v : \mathscr{R}_{\mathscr{F}}(M,N) \to N$ and $\mathscr{R}_{\mathscr{F}}$ is a hofunctor that is independent of M and N. The value of $\mathscr{R}_{\mathscr{F}}(M,N)$ and the types of the parameters in $\overline{v}_{\mathscr{F}}$ are determined by the following inductive definition.

$$\begin{array}{llcll}
\text{(a)} & \Phi_{\mathscr{K}_Q}\;\langle\rangle\; u & = & id_{Q\cdot M} & \\
\text{(b)} & \Phi_{\mathscr{I}d}\;\langle r\rangle\; u & = & u \cdot T\, r & r : M \to M \\
\text{(c)} & \Phi_{\mathscr{F}\ddot{+}\mathscr{G}}\;(\overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}})\; u & = & \Phi_{\mathscr{F}}\;\overline{v}_{\mathscr{F}}\; u + \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u & \\
\text{(d)} & \Phi_{\mathscr{F}\ddot{\times}\mathscr{G}}\;(\overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}})\; u & = & \Phi_{\mathscr{F}}\;\overline{v}_{\mathscr{F}}\; u \times \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u & \\
\text{(e)} & \Phi_{\mathscr{K}_Q\dot{\star}\mathscr{G}}\;(\langle s\rangle \frown \overline{v}_{\mathscr{G}})\; u & = & Q\,(s \cdot \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u) & s \cdot \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u : \mathscr{G}T \cdot M \to M \cdot \mathscr{G}N \\
\text{(f)} & \Phi_{\mathscr{I}d\dot{\star}\mathscr{G}}\;(\langle t\rangle \frown \overline{v}_{\mathscr{G}})\; u & = & u_{\mathscr{G}N} \cdot T(\,t \cdot \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u) & t \cdot \Phi_{\mathscr{G}}\;\overline{v}_{\mathscr{G}}\; u : \mathscr{G}T \cdot M \to M \cdot \mathscr{G}N
\end{array}$$

where $s = id_M$ if $\mathscr{G} = \mathscr{K}_{Id}$.

6.1. Examples of generalised folds

First we will show how the generalised fold on the datatype *Nest* given in Section 3 can be derived from the inductive definition above. Then we will give one more example: the generalised fold on the datatype *Bush*. Two further examples are given in [BP99b].

6.1.1. *Nest*

The *Nest* datatype is an initial algebra of the hofunctor $\mathscr{N}$ defined in equation (16) by

$$\mathscr{N} \quad = \quad \mathscr{K}_{K_1} \ddot{+} \mathscr{K}_{Id} \ddot{\times} \mathscr{I}d \dot{\star} \mathscr{K}_{Pair}$$

So we can calculate that the parameter collection $\overline{v}_{\mathscr{N}}$ contains only one parameter t, and

$$\begin{array}{ll}
 & \Phi_{\mathscr{N}}\;\langle t\rangle\; u \\
= & \quad \text{definition of } \mathscr{N} \\
 & \Phi_{\mathscr{K}_{K_1}\ddot{+}\mathscr{K}_{Id}\ddot{\times}\mathscr{I}d\dot{\star}\mathscr{K}_{Pair}}\;\langle t\rangle\; u \\
= & \quad \text{(a), (c) and (d)} \\
 & \Phi_{\mathscr{K}_{K_1}}\;\langle\rangle\; u + \Phi_{\mathscr{K}_{Id}}\;\langle\rangle\; u \times \Phi_{\mathscr{I}d\dot{\star}\mathscr{K}_{Pair}}\;\langle t\rangle\; u \\
= & \quad \text{(a) and (f)} \\
 & id_{K_1} + id_M \times u_{Pair} \cdot nest\,(t \cdot \Phi_{\mathscr{K}_{Pair}}\langle\rangle\; u) \qquad\qquad t : Pair \cdot M \to M \cdot Pair \\
= & \quad \text{(a)} \\
 & id_{K_1} + id_M \times u_{Pair} \cdot nest\; t
\end{array}$$

We can deduce the type of parameter v of equation (17) from the type of $\Phi_{\mathscr{N}}\;\langle t\rangle\; u$:

$$v : 1 + M \times N \cdot Pair \to N$$

If we write v as a coproduct $[e,f]$, take $t = g$ and suppose that $\overline{v}$ represents the parameters e, f and g, then

$$gfold_{\mathscr{F}}\ \overline{v} \cdot \alpha \quad = \quad [e,f] \cdot (id_{K_1} + id_M \times (gfold_{\mathscr{F}}\ \overline{v}) \cdot nest\ g)$$

which corresponds to the Haskell definition of the generalised fold in equation (6) of Section 3.

6.1.2. Bush

The datatype *Bush* was introduced in [BM98]. Its definition is the following:

$$\textbf{data}\ Bush\ a \quad = \quad Nil \mid (Cons\ (a,\ Bush\,(Bush\ a))$$

This datatype is the initial algebra of the following hofunctor:

$$\mathscr{B} \quad = \quad \mathscr{K}_{K_1} \ddot{+} \mathscr{K}_{Id} \ddot{\times} \mathscr{I}d \dot{\star} \mathscr{I}d \tag{18}$$

Therefore the map function for *Bush* is defined by:

$$\begin{array}{lcl} \multicolumn{3}{l}{bush :: \forall a.\forall b.\ (a \to b) \to Bush\ a \to Bush\ b} \\ bush\,f\ Nil & = & Nil \\ bush\,f\ (Cons\ (x,xs)) & = & Cons\ (f\ x,\ bush\,(bush\,f)\ xs) \end{array}$$

and the parameter collection $\overline{v}_{\mathscr{B}} = \langle r,\ t\rangle$ for some r and t. The calculation of $\Phi_{\mathscr{B}}\ \langle r,\ t\rangle\ u$ is similar to that of $\Phi_{\mathscr{N}}\ \langle t\rangle\ u$ given above, so some of the steps are omitted:

$$\begin{array}{cll} & \Phi_{\mathscr{B}}\ \langle r,\ t\rangle\ u & \\ = & \quad \text{(a), (c), (d) and (f)} & \\ & id_{K_1} + id_M \times u_N \cdot bush(t \cdot \Phi_{\mathscr{I}d}\ \langle r\rangle\ u) & t : N \to M \cdot N \\ = & \quad \text{(b)} & \\ & id_{K_1} + id_M \times u_N \cdot bush(t \cdot u \cdot bush\ r) & r : M \to M \end{array}$$

So this time the final parameter v has type

$$v : 1 + M \times N \cdot N \to N$$

If we write v as a coproduct $[e,f]$, take $t = g$ and $r = h$ and suppose that $\overline{v}$ represents the parameters e, f, g and h, then we have

$$gfold_{\mathscr{F}}\ \overline{v} \cdot \alpha \quad = \quad [e,f] \cdot (id_{K_1} + id_M \times (gfold_{\mathscr{F}}\ \overline{v}) \cdot bush(g \cdot (gfold_{\mathscr{F}}\ \overline{v}) \cdot bush\ h))$$

which corresponds to the following Haskell definition:

$$\begin{array}{lcl} \multicolumn{3}{l}{gfold :: \forall n.\forall m.\forall b.\ (\forall a.\ n\ a) \to (\forall a.\ (m\ a, n(n\ a)) \to n\ a) \to (\forall a.\ n\ a \to m(n\ a)) \to (\forall a.\ m\ a \to m\ a)} \\ \multicolumn{3}{l}{\qquad\qquad \to Bush\ (m\ b) \to n\ b} \\ gfold\ e\ f\ g\ h\ Nil & = & e \\ gfold\ e\ f\ g\ h\ (Cons\ (x,xs)) & = & f\ (x, (gfold\ e\ f\ g\ h \cdot bush\ (g \cdot gfold\ e\ f\ g\ h \cdot bush\ h))\ xs) \end{array}$$

The ordinary fold on bushes is then a special case of this definition, when specialised to the appropriate type:

$$\begin{array}{lcl} \multicolumn{3}{l}{fold :: \forall a.\forall b.\ a \to ((a,a) \to a) \to (a \to a) \to (a \to a) \to Bush\ a \to a} \\ fold\ e\ f & = & gfold\ e\ f\ id\ id \end{array}$$

Unlike the ordinary fold on nests, this fold can be used to sum a bush, but its restrictive type $Bush\ a \to a$ still makes it rather inflexible.

6.2. Comparison with Hinze's generalised fold

Before introducing our definition of an efficient fold, we pause to examine the difference between the generalised folds of [BP99b] and [Hin00], since there are corresponding differences between the efficient folds.

The first difference is in the right hand side of clause (b) in the definition of Φ at the start of Section 6, which would become simply u in Hinze's definition. One result of this change is that Hinze's generalised folds coincide with ordinary folds on regular datatypes. Bird and Paterson made the observation that their definition could have been changed in this way, but chose to keep the extra parameter for maximum generality. The initial algebra characterisation of both generalised folds and the corresponding efficient folds would still be valid if this change were made.

There are more fundamental differences in clauses (e) and (f), and in the type signatures of the parameters. These differences are perhaps best illustrated on the above examples. The difference for *Nest* is only in the type signature, as we observed in Section 3, but we can see a much more marked difference by considering Hinze's definition for *Bush*:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{gfoldB} :: \forall n.\forall m.\forall b.\ (\forall a.\ n\,a) \to (\forall a.\ (m\,a, n(n\,a)) \to n\,a) \to (\forall a.\ m(n\,a))} \\
\multicolumn{3}{l}{\qquad\qquad \to (\forall a.\ (m\,a, m\,(n\,(n\,a))) \to m\,(n\,a)) \to \mathit{Bush}\,(m\,b) \to n\,b} \\
\mathit{gfoldB}\,e\,f\,g\,h\,\mathit{Nil} & = & e \\
\mathit{gfoldB}\,e\,f\,g\,h\,(\mathit{Cons}\,(x,xs)) & = & f\,(x,\,(\mathit{gfoldB}\,e\,f\,g\,h\cdot \mathit{bush}\,(\mathit{kfoldB}\,g\,h\,))\,xs)
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{kfoldB} :: \forall n.\forall m.\forall b.\ (\forall a.\ m(n\,a)) \to (\forall a.\ (m\,a, m\,(n\,(n\,a))) \to m\,(n\,a)) \to \mathit{Bush}\,(m\,b) \to n\,b} \\
\mathit{kfoldB}\,g\,h\,\mathit{Nil} & = & g \\
\mathit{kfoldB}\,g\,h\,(\mathit{Cons}\,(x,xs)) & = & h\,(x,\,(\mathit{kfoldB}\,g\,h\cdot \mathit{bush}\,(\mathit{kfoldB}\,g\,h\,))\,xs)
\end{array}$$

Note that *kfoldB* has the same definition as an ordinary fold on *Bush*, but has a different type signature. This definition is clearly very different from Bird and Paterson's, as presented in Section 6.1.2.

7. An Efficient Fold

We will now give the definition of the efficient fold. The definition is given inductively, like that of the generalised fold.

Let $\mathscr{F} : \mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ be a nested hofunctor with least fixed point T and initial algebra $\alpha : \mathscr{F}T \to T$, and let $\overline{v} = \langle v\rangle \frown \overline{v}_{\mathscr{F}}$ for some v, where the parameter collection $\overline{v}_{\mathscr{F}}$ has the same definition as in Section 6. Then, given functors M and N, the efficient fold $\mathit{efold}_{\mathscr{F}}\ \overline{v}$ has type

$$\mathit{efold}_{\mathscr{F}}\ \overline{v} : (\forall L,X.\ (L \to M\cdot X) \to T\cdot L \to N\cdot X)$$

and is characterised uniquely (see Theorem 7.2 below) by the universal property that for all $u : (\forall L,X.\ (L \to M\cdot X) \to T\cdot L \to N\cdot X)$

$$u = \mathit{efold}_{\mathscr{F}}\,\overline{v} \quad \Leftrightarrow \quad (\forall p : L \to M\cdot X : u\,p\cdot\alpha_L = v_X\cdot\Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\,p) \tag{19}$$

where

$$\Theta_{\mathscr{F}}\,\overline{v}_{\mathscr{F}} : (\forall L,X.\ (L \to M\cdot X) \to T\cdot L \to N\cdot X) \to (L' \to M\cdot X') \to \mathscr{F}T\cdot L' \to \mathscr{R}_{\mathscr{F}}(M,N)\cdot X'$$

and where $v : \mathscr{R}_{\mathscr{F}}(M,N) \to N$, and v_X denotes $v\star id_X$. The types given to the parameters in $\overline{v}_{\mathscr{F}}$ in the following definition of Θ are the same as for generalised folds, as is the resulting value of $\mathscr{R}_{\mathscr{F}}$.

$$\begin{array}{llcll}
\text{(a)} & \Theta_{\mathscr{K}_Q}\ \langle\rangle\ u\,p & = & Q\,p & \\
\text{(b)} & \Theta_{Id}\ \langle r\rangle\ u\,p & = & u\,(r_{X'}\cdot p) & r :: M \to M \\
\text{(c)} & \Theta_{\mathscr{F}\dot{\ddot{+}}\mathscr{G}}\ (\overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}})\ u\,p & = & \Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\,p + \Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p & \\
\text{(d)} & \Theta_{\mathscr{F}\ddot{\times}\mathscr{G}}\ (\overline{v}_{\mathscr{F}} \frown \overline{v}_{\mathscr{G}})\ u\,p & = & \Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\,p \times \Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p & \\
\text{(e)} & \Theta_{\mathscr{K}_Q\dot{\star}\mathscr{G}}\ (\langle s\rangle \frown \overline{v}_{\mathscr{G}})\ u\,p & = & Q\,(s_{X'}\cdot\Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p) & s_{X'}\cdot\Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p :: \mathscr{G}T\cdot L' \to M\cdot\mathscr{G}N\cdot X' \\
\text{(f)} & \Theta_{\mathscr{I}d\dot{\star}\mathscr{G}}\ (\langle t\rangle \frown \overline{v}_{\mathscr{G}})\ u\,p & = & u\,(t_{X'}\cdot\Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p) & t_{X'}\cdot\Theta_{\mathscr{G}}\ \overline{v}_{\mathscr{G}}\ u\,p :: \mathscr{G}T\cdot L' \to M\cdot\mathscr{G}N\cdot X'
\end{array}$$

where $s = id_M$ if $\mathscr{G} = \mathscr{K}_{Id}$. The relationship between Θ and its counterpart for generalised folds, Φ, is captured by the following lemma which will be used later in the proof of Theorem 7.3:

Lemma 7.1. Let $\mathscr{F} : \mathbf{Coc}(\mathbf{C}) \to \mathbf{Coc}(\mathbf{C})$ be a nested hofunctor with least fixed point T, and let $\overline{v}_{\mathscr{F}}$ be a parameter collection of appropriate type. Suppose that $y = \mathit{gfold}_{\mathscr{F}}(\langle v\rangle \frown \overline{v}_{\mathscr{F}})$ for some v, and let $u : (\forall L,X.\ (L \to M\cdot X) \to T\cdot L \to N\cdot X)$ be defined by

$$u\,p = y_X\cdot T\,p$$

then for all L, M and X, and all $p : L \to M\cdot X$,

$$\Theta_{\mathscr{F}}\,\overline{v}_{\mathscr{F}}\ u\,p \quad = \quad (\Phi_{\mathscr{F}}\,\overline{v}_{\mathscr{F}}\ y)_X\cdot\mathscr{F}\ T\,p$$

The proof of this lemma is omitted, since it consists of routine case analysis on each of the clauses (a) to (f) in the definitions of Φ and Θ.

7.1. Examples of efficient folds

The parameter collections for both the following examples are the same as those given in Section 6.1 for the corresponding generalised folds.

7.1.1. Nest

Using the definition of $\mathscr{N}$ in equation (16) we calculate:

$$\begin{array}{ll}
 & \Theta_{\mathscr{N}}\ \langle t\rangle\ u\ p \\
= & \quad \text{definition of } \mathscr{N} \\
 & \Theta_{\mathscr{K}_{K_1} \dot{+} \mathscr{K}_{Id} \ddot{\times} \mathscr{I}d \dot{\star} \mathscr{K}_{Pair}}\ \langle t\rangle\ u\ p \\
= & \quad \text{(a), (c) and (d)} \\
 & \Theta_{\mathscr{K}_{K_1}}\ \langle\rangle\ u\ p + \Theta_{\mathscr{K}_{Id}}\ \langle\rangle\ u\ p \times \Theta_{\mathscr{I}d \dot{\star} \mathscr{K}_{Pair}}\ \langle t\rangle\ u\ p \\
= & \quad \text{(a)} \\
 & id_{K_1} + p \times \Theta_{\mathscr{I}d \dot{\star} \mathscr{K}_{Pair}}\ \overline{w}\ u\ p \\
= & \quad \text{(a)} \\
 & id_{K_1} + p \times u\ (t_{X'} \cdot \Theta_{\mathscr{K}_{Pair}} \langle\rangle\ u\ p) \qquad t_{X'} \cdot \Theta_{\mathscr{K}_{Pair}} \langle\rangle\ u\ p : Pair \cdot L \rightarrow M \cdot Pair \cdot X \\
= & \quad \text{(a)} \\
 & id_{K_1} + p \times u\ (t_{X'} \cdot pair\ p) \qquad t : Pair \cdot M \rightarrow M \cdot Pair
\end{array}$$

If $v = [e,f]$, $t = g$ and $p = h$ and if we suppose that $\overline{v}$ represents the parameters e, f and g, then we have

$$efold_{\mathscr{F}}\ \overline{v}\ h \cdot \alpha \quad = \quad [e,f] \cdot (id_{K_1} + h \times (efold_{\mathscr{F}}\ \overline{v}\ (g \cdot pair\ h)))$$

which corresponds to the Haskell definition of the efficient fold in equation 12 of Section 3.

7.1.2. Bush

We can use the definition of $\mathscr{B}$ in equation (18) to calculate

$$\begin{array}{ll}
 & \Theta_{\mathscr{B}}\ \langle r,t\rangle\ u\ p \\
= & \quad \text{(a), (c), (d) and (f)} \\
 & id_{K_1} + p \times u\ (t_{X'} \cdot \Theta_{\mathscr{I}d}\ \langle r\rangle\ u\ p) \qquad t_{X'} \cdot \Theta_{\mathscr{I}d}\ \langle r\rangle\ u\ p : T \cdot L' \rightarrow M \cdot N \cdot X' \\
= & \quad \text{(b)} \\
 & 1 + p \times u\ (t_{X'} \cdot (u\ (r_{X'} \cdot p))) \qquad r : M \rightarrow M,\ t : N \rightarrow M \cdot N
\end{array}$$

Taking $v = [e,f]$, $t = g$, $r = h$ and $p = k$ gives the definition:

$$\begin{array}{lcl}
\multicolumn{3}{l}{efold :: \forall n.\forall m.\forall b.(\forall a.\ n\ a) \rightarrow (\forall a.\ (m\ a, n\ (n\ a)) \rightarrow n\ a) \rightarrow (\forall a.\ n\ a \rightarrow m\ (n\ a)) \rightarrow (\forall a.\ m\ a \rightarrow m\ a)} \\
\multicolumn{3}{l}{\qquad\qquad \rightarrow (l\ b \rightarrow m\ (z\ b)) \rightarrow Bush\ (l\ b) \rightarrow n\ (z\ b)))} \\
efold\ e\ f\ g\ h\ k\ Nil & = & e \\
efold\ e\ f\ g\ h\ k\ (Cons\ (x,xs)) & = & f\ (k\ x, efold\ e\ f\ g\ h\ (g \cdot (efold\ e\ f\ g\ h\ (h \cdot k)))\ xs)
\end{array}$$

Comparing this definition with the corresponding one for the generalised fold, we can easily establish the familiar relationship that for all $e : \forall a.\ n\ a$, $f : \forall a.\ (m\ a, n\ (n\ a)) \rightarrow n\ a$, $g : \forall a.\ n\ a \rightarrow m\ (n\ a)$, $h : \forall a.\ m\ a \rightarrow m\ a$ and $k : \forall a. l\ a \rightarrow m\ (z\ a)$

$$efold\ e\ f\ g\ h\ k = gfold\ e\ f\ g\ h \cdot bush\ k$$

This result is proved for arbitrary datatypes in Theorem 7.3 of the next section. The efficient fold of [Hin00] was specified similarly as $gfoldB\ e\ f\ g\ h \cdot bush\ k$, where $gfoldB$ was defined in Section 6.2. The differences between the two generalised folds show that this would lead to a very different efficient fold.

7.2. Uniqueness of efficient folds

We will now show that efficient folds are characterised uniquely by their defining equations. The corresponding property of generalised folds is deduced from a more general result in [BP99b], which is stated in Theorem 7.1 below. We will use the notation

$$A \rightarrow_{\mathbf{C}} B$$

to denote the (hom)set of all arrows from A to B in $\mathbf{C}$. The theorem below uses the contravariant hom-functor, which we will write as $\_\_ \rightarrow_{\mathbf{C}} B$ for each object B in $\mathbf{C}$. It sends each object A to the set $A \rightarrow_{\mathbf{C}} B$ and each arrow $g : A \rightarrow A'$ to the function $\lambda f. f \cdot g$.

Theorem 7.1 ([BP99b]). Suppose that $F :: \mathbf{C} \rightarrow \mathbf{C}$ has initial algebra $\alpha : F\ T \rightarrow T$, and

$$\Psi :: \forall A.\ (P\ A \rightarrow_{\mathbf{D}} B) \rightarrow (P(F\ A) \rightarrow_{\mathbf{D}} B)$$

is a natural transformation for some functor $P : \mathbf{C} \rightarrow \mathbf{D}$. Then if F and P preserve colimits of chains and P preserves initiality, there is a unique $x :: P\ T \rightarrow B$ such that

$$x \cdot P\alpha \quad = \quad \Psi\, x. \tag{20}$$

Fortunately the following theorem shows that the uniqueness theorem for efficient folds is also an instance of Theorem 7.1. So this justifies definition (19).

Theorem 7.2. Let $\mathscr{F} : \mathbf{Coc}(\mathbf{C}) \rightarrow \mathbf{Coc}(\mathbf{C})$ be a nested hofunctor with initial algebra $\alpha : \mathscr{F}\ T \rightarrow T$, and let $\overline{v} = \langle v \rangle ^\frown \overline{v}_{\mathscr{F}}$ be a parameter collection of appropriate type. Then there is a unique $u : \forall L, X.\ (L \rightarrow_{\mathbf{Coc}(\mathbf{C})} M \cdot X) \rightarrow T \cdot L \rightarrow_{\mathbf{Coc}(\mathbf{C})} N \cdot X$ such that for all $p : L \rightarrow M \cdot X$

$$u\ p \cdot \alpha_L = v_X \cdot \Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\ p. \tag{21}$$

Proof. Let the category $\mathbf{D}$ be defined as follows:

- The objects are functions which map pairs of endofunctors on $\mathbf{Coc}(\mathbf{C})$ to a single endofunctor. So, if R is an object of $\mathbf{D}$, and $L, X : \mathbf{C} \rightarrow \mathbf{C}$ are functors, then $R(L,X) : \mathbf{C} \rightarrow \mathbf{C}$ is a functor.
- The arrows are natural transformations $k : R \rightarrow S$ in $\mathbf{D}$ of type
  $$k : (\forall L, X.\ (L \rightarrow_{\mathbf{Coc}(\mathbf{C})} M \cdot X) \rightarrow R(L,X) \rightarrow_{\mathbf{Coc}(\mathbf{C})} S(L,X)).$$
- The composition $\circ$ of $k : R \rightarrow S$ and $l : S \rightarrow T$ in $\mathbf{D}$ is defined by:
  $$l \circ k \quad = \quad \lambda p.(l\ p \cdot k\ p)$$
  where $\cdot$ denotes composition in $\mathbf{Coc}(\mathbf{C})$.
- The identity of this composition is then defined for each object R by $id_R\ (L,X) = \lambda p.\, id_{R(L,X)}$, where $id_{R(L,X)}$ denotes the identity of the functor $R(L,X)$ in $\mathbf{Coc}(\mathbf{C})$.

Now define $\mathscr{P} : \mathbf{Coc}(\mathbf{C}) \rightarrow \mathbf{D}$ on each object F and arrow $\gamma : F \rightarrow G$ by

$$\begin{aligned} \mathscr{P}\ F\ (L,X) &\quad = \quad F \cdot L \\ \mathscr{P}\ \gamma\ (L,X) &\quad = \quad \lambda p. \gamma_L. \end{aligned}$$

Then let $B(L,X) = N \cdot X$, and define

$$\begin{aligned} &\Pi_{\mathscr{F}} :: \forall A.\ (\mathscr{P}\ A \rightarrow_{\mathbf{D}} B) \rightarrow (\mathscr{P}(\mathscr{F}\ A) \rightarrow_{\mathbf{D}} B) \\ &\Pi_{\mathscr{F}}\ \overline{v}\ u\ p = v_X \cdot \Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\ p \end{aligned}$$

for all $u : (\forall L, X.\ (L \rightarrow M \cdot X) \rightarrow T \cdot L \rightarrow N \cdot X)$ and $p : L \rightarrow M \cdot X$. Then $\Pi_{\mathscr{F}}$ is a natural transformation, so statement (21) follows by Theorem 7.1 provided that $\mathscr{P}$ preserves initiality and colimits of chains. These properties may be verified using the fact that limits and colimits in $\mathbf{Coc}(\mathbf{C})$ are calculated pointwise from those in $\mathbf{C}$ [Mac98]. $\square$

Theorem 7.2 can be used to prove the following:

Theorem 7.3. Suppose that $\mathscr{F} : \mathbf{Coc}(\mathbf{C}) \rightarrow \mathbf{Coc}(\mathbf{C})$ has initial algebra $\alpha : \mathscr{F}\ T \rightarrow T$. Then for all collections of parameters $\overline{v}$ of appropriate type, and all $p : L \rightarrow M \cdot X$,

$$efold_{\mathscr{F}}\ \overline{v}\ p \quad = \quad (gfold_{\mathscr{F}}\ \overline{v})_X \cdot T\ p.$$

Proof Outline Suppose $\overline{v} = \langle v \rangle ^\frown \overline{v}_{\mathscr{F}}$, let $y = gfold_{\mathscr{F}}\ \overline{v}$ and define $u : (\forall L, X.\ (L \rightarrow M \cdot X) \rightarrow T \cdot L \rightarrow N \cdot X)$ by

$$u\ p = y_X \cdot T\ p$$

By the universal property of efficient folds (19)

$$u = efold_{\mathscr{F}}\ \overline{v} \quad \Leftrightarrow \quad (\forall p : L \rightarrow M \cdot X : u\ p \cdot \alpha_L = v_X \cdot \Theta_{\mathscr{F}}\ \overline{v}_{\mathscr{F}}\ u\ p)$$

We can calculate that

$$\begin{array}{ll}
 & u\, p \cdot \alpha_L \\
= & \quad \text{definition of } u \\
 & y_X \cdot T\, p \cdot \alpha_L \\
= & \quad \text{naturality of } \alpha \\
 & y_X \cdot \alpha_{M \cdot X} \cdot \mathscr{F}\; T\, p \\
= & \quad \text{universal property of generalised folds (17)} \\
 & v_X \cdot (\Phi_{\mathscr{F}}\, \overline{v}_{\mathscr{F}}\; y)_X \cdot \mathscr{F}\; T\, p \\
= & \quad \text{Lemma 7.1} \\
 & v_X \cdot \Theta_{\mathscr{F}}\, \overline{v}_{\mathscr{F}}\; u\, p)
\end{array}$$

which establishes the result. □

7.3. Fusion laws for efficient folds

The following map fusion law is immediate from Theorem 7.3 and the functorality of T.

Law 7.1 (Map fusion I). Let $k : L' \to L$, then for all $p : L \to M \cdot X$

$$efold_{\mathscr{F}}\; \overline{v}\; p \cdot T\; k = efold_{\mathscr{F}}\; \overline{v}\; (p \cdot k)$$

The fusion laws of [BP99b], together with Theorem 7.3 give two further fusion laws for efficient folds:

Law 7.2 (Map fusion II). Suppose that for each $i \in 0..m$,

$$v_i :: \mathscr{P}_i M \to \mathscr{Q}_i M \text{ and } v_i' :: \mathscr{P}_i L \to \mathscr{Q}_i L$$

where $\mathscr{P}_i$ and $\mathscr{Q}_i$ are hofunctors that are independent of M. Then if $p : L \to M \cdot X$

$$(\forall i \in 1..m : v_i \cdot \mathscr{P}_i p = \mathscr{Q}_i p \cdot v_i') \Rightarrow efold_{\mathscr{F}}\; \overline{v}\; p = efold_{\mathscr{F}}\; \overline{w}\; id$$

where $w_0 = v_0 \cdot \mathscr{R}_{\mathscr{F}}(p, id_N)$ and for all $i \in 1..m$, $w_i = v_i'$.

Law 7.3 (Fold fusion). Let $k : N \to N'$, and suppose that for each $i \in 0..m$,

$$v_i :: \mathscr{P}_i N \to \mathscr{Q}_i N \text{ and } v_i' :: \mathscr{P}_i N' \to \mathscr{Q}_i N'$$

where $\mathscr{P}_i$ and $\mathscr{Q}_i$ are hofunctors that are independent of N. Then

$$(\forall i \in 0..m : \mathscr{Q}_i k \cdot v_i = v_i' \cdot \mathscr{P}_i k) \Rightarrow k \cdot efold_{\mathscr{F}}\; v_0 \cdots v_m = efold_{\mathscr{F}}\; v_0' \cdots v_m'.$$

This fold fusion law is an instance of a stronger law given for three specific types in [BP99b], and given here for nests in (10). The weaker law for nests, corresponding to the above, is obtained by taking k' to be the identity function in (10). The stronger law cannot be simply stated for arbitrary datatypes, so the reader is referred to [BP99b] for details.

8. Applications

The fusion laws of the previous section are perhaps a bit obscure until applied to a concrete example. We will now use them to give conditions under which an efficient fold can be rewritten as an ordinary fold for a small class of nested datatypes. We will then apply this result to functions on two different datatypes: nests and lists. We will show that there are two equivalent ways to use efficient folds to flatten a nest to a list. The two methods correspond to the inverse methods of building a tree: recursive, or top-down, and iterative, or bottom-up. The application to lists concerns Horner's rule for evaluating polynomials [BdM97]. We will show that this is a trivial consequence of the fusion laws.

We will only consider datatypes of the form:

$$\textbf{data}\; Collection\; a \quad = \quad Nil \mid Cons\; (a, Collection\; (Fun\; a))$$

where Fun is a functor. So for example, if $Fun = Pair$, then $Collection$ defines nests, and if $Fun = Id$ it defines lists.

The efficient fold on type *Collection* is identical to that on nests, except that *Pair* is generalised to an arbitrary functor *Fun*:

$$\begin{array}{lcl} \mathit{efold} :: \forall n.\forall m.\forall b.(\forall a.\ n\ a) \to (\forall a.\ (m\ a, n\ (\mathit{Fun}\ a)) \to n\ a) \to (\forall a.\ \mathit{Fun}\ (m\ a) \to m\ (\mathit{Fun}\ a)) \to \\ \qquad\qquad (\forall l.(l\ a \to m\ b) \to \mathit{Collection}\ (l\ a) \to n\ b) \\ \mathit{efold}\ e\ f\ g\ h\ \mathit{Nil} & = & e \\ \mathit{efold}\ e\ f\ g\ h\ (\mathit{Cons}\ (x, xs)) & = & f\ (h\ x, \mathit{efold}\ e\ f\ g\ (g \cdot \mathit{fun}\ h)\ xs) \end{array}$$

The fusion laws are also very similar to those for nests, but are restated here for easy reference in the proof of Law 8.4 below. The fold fusion law has one less parameter than its counterpart for nests (10) because it is the weaker version of the law, as given in Section 7.3. Assuming the types given to parameters *e*, *f*, *g* and *h* in the definition of the efficient fold above, the fold and map fusion laws of Section 7.3 can be instantiated to the following, where *coll* represents the map function for the datatype *Collection*.

Law 8.1 (Map fusion I). Let $k' : \forall a.\ l'\ a \to l\ a$, then

$$\mathit{efold}\ e\ f\ g\ h \cdot \mathit{coll}\ k = \mathit{efold}\ e\ f\ g\ (h \cdot k)$$

Law 8.2 (Map fusion II). Let $g' : (\forall a.\ \mathit{Fun}\ (l\ a) \to l\ (\mathit{Fun}\ a))$, then

$$\begin{array}{ll} & g \cdot \mathit{fun}\ h = h \cdot g' \\ \Rightarrow & \\ & \mathit{efold}\ e\ f\ g\ h = \mathit{efold}\ e\ (f \cdot (h \times id))\ g'\ id \end{array}$$

Law 8.3 (Fold fusion). Let $e' : \forall a.\ n'\ a$, $f' : \forall a.\ m\ a \to n'\ (\mathit{Fun}\ a) \to n'\ a$ and $k : \forall a.\ n\ a \to n'\ a$, then

$$\begin{array}{ll} & k\ e = e' \text{ and } k \cdot f = f' \cdot (id \times k) \\ \Rightarrow & \\ & k \cdot \mathit{efold}\ e\ f\ g\ h = \mathit{efold}\ e'\ f'\ g\ h \end{array}$$

These laws can be used together to deduce the following law:

Law 8.4 (Fold Equivalence). Let $e : \forall a.\ n\ a$, $f : (m\ a, n\ a) \to n\ a$, $g : \mathit{Fun}\ (m\ a) \to n\ a$, $h : \forall a.\ a \to n\ a$ and suppose there exist $k : n\ (\mathit{Fun}\ a) \to n\ a$ and $k' : m\ (\mathit{Fun}\ a) \to m\ a$, then

$$\begin{array}{ll} & k\,e = e \quad k \cdot f = f \cdot (k' \times k) \quad g \cdot \mathit{fun}\ k' = k' \cdot g \ \text{ and } \ g \cdot \mathit{fun}\ h = k' \cdot h \\ \Rightarrow & \\ & \mathit{efold}\ e\ f\ g\ h = \mathit{fold}\ e\ (f \cdot (h \times k)) \end{array}$$

Proof. We have

$$\begin{array}{ll} & \mathit{efold}\ e\ f\ g\ h = \mathit{fold}\ e\ (f \cdot (h \times k)) \\ \Leftrightarrow & \quad \text{universal property of } \mathit{fold} \\ & \mathit{efold}\ e\ f\ g\ h\ \mathit{Nil} = e \text{ and } \mathit{efold}\ e\ f\ g\ h\ (\mathit{Cons}\ (x, xs)) = f\ (h\ x, (k \cdot \mathit{efold}\ e\ f\ g\ h)\ xs) \\ \Leftarrow & \quad \text{definition of } \mathit{efold} \\ & \mathit{efold}\ e\ f\ g\ (g \cdot \mathit{fun}\ h) = k \cdot \mathit{efold}\ e\ f\ g\ h \end{array}$$

This equality is established by appealing to the fusion laws:

$$\begin{aligned}
& k \cdot \mathit{efold}\ e\ f\ g\ h \\
= \quad & \text{Law 8.3 (fold fusion) and functorality of } \times \\
& \mathit{efold}\ e\ (f \cdot (k' \times id))\ g\ h \\
= \quad & \text{Law 8.1 (map fusion I)} \\
& \mathit{efold}\ e\ (f \cdot (k' \times id))\ g\ id \cdot \mathit{coll}\ h \\
= \quad & \text{Law 8.2 (map fusion II)} \\
& \mathit{efold}\ e\ f\ g\ k' \cdot \mathit{coll}\ h \\
= \quad & \text{Law 8.1 (map fusion I)} \\
& \mathit{efold}\ e\ f\ g\ (k' \cdot h) \\
= \quad & g \cdot \mathit{fun}\ h = k' \cdot h \\
& \mathit{efold}\ e\ f\ g\ (g \cdot \mathit{fun}\ h)
\end{aligned}$$

□

We will give three applications of this law: two for nests and one for lists. The two functions on nests are *flatten* and *size*, which were cited in Section 3 as examples for which the map fusion law (11) did not directly apply. Writing these functions as efficient folds, we have

$$\begin{aligned}
\mathit{flatten} &= \mathit{efold}\ []\ (\mathit{uncurry}\ (+\!\!+))\ (\mathit{uncurry}\ (+\!\!+))\ (\lambda x.\ [x]) \\
\mathit{size} &= \mathit{efold}\ 0\ (\mathit{uncurry}\ (+))\ (\mathit{uncurry}\ (+))\ (\lambda x.\ 1)
\end{aligned}$$

The *flatten* function flattens a nest by a method that corresponds to the inverse method of recursively building a tree, but the fold equivalence law transforms it into a function corresponding to the inverse method of iteratively building a tree: if $m\ a = n\ a = \mathit{List}\ a$, and $k = k' = \mathit{concat} \cdot \mathit{map}\ (\lambda(y,z).\ [y,z])$, then

$$\mathit{flatten} = \mathit{fold}\ []\ ((\mathit{uncurry}\ (+\!\!+)) \cdot ((\lambda x.\ [x]) \times k))$$

For the *size* function, we take $m\ a = n\ a = \mathit{Int}$, and $k = k' = (2*)$, which gives

$$\mathit{size} = \mathit{fold}\ 0\ ((\mathit{uncurry}\ (+))\ ((\lambda x.\ 1) \times (2*)))$$

Finally, we notice that Horner's rule for lists [BdM97] can be derived very simply by taking $m = n$, $k = k' = g$ and $h = id$, which leaves only two conditions:

$$\begin{aligned}
& g\ e = e \text{ and } g \cdot f = f \cdot (g \times g) \\
\Rightarrow \quad & \\
& \mathit{efold}\ e\ f\ g\ id = \mathit{fold}\ e\ (f \cdot (id \times g))
\end{aligned}$$

By applying this law to the classic example of polynomial evaluation, we can see that it is a restatement of Horner's rule: let $us = \mathit{Cons}\ (u,\ \mathit{Cons}\ (v,\ \mathit{Cons}\ (w,\ \mathit{Nil})))$, then

$$\begin{aligned}
\mathit{efold}\ 0\ (+)\ (*x)\ id\ us &= u + v*x + w*x^2 \\
\mathit{fold}\ 0\ (\mathit{apply}\ (+)\ id\ (*x))\ us &= u + (v + (w+0)*x)*x
\end{aligned}$$

Horner's rule was generalised to arbitrary regular datatypes in [BdM97]; whether there is a similar generalisation for nested datatypes remains to be seen.

9. Conclusion

In this paper we have given a universal construction of a fold operator that was inspired by that of [Hin00]. This construction comes equipped with a proof principle that we have used to relate efficient folds to generalised folds on arbitrary nested datatypes in Theorem 7.3. The performance improvement outlined in [Hin00] has been reiterated here through the example of random access lists, and the fusion laws of [BP99b] have been restated in the context of efficient folds. The laws have then been used to derive a fold equivalence law that gives conditions that are sufficient to rewrite an efficient fold as an ordinary fold. The fold equivalence law has yet to be generalised to arbitrary datatypes.

We have observed that folds in general provide a useful definition principle for functions on nested datatypes. Without them it is difficult to define even a function as simple as *sum*. We have noted that efficient folds are not always more efficient than generalised ones, but they do capture a different pattern of computation. Furthermore, they are arguably more closely related than generalised folds to ordinary folds, because they have a similar map fusion law. Although the applications of nested datatypes have so far been quite limited, the example of Horner's rule in Section 8 suggests that the proof rules associated with efficient folds could still be useful for program derivation on regular datatypes. It might be interesting to investigate whether there is a class of problems on regular datatypes that can be specified as efficient folds, and then manipulated through the proof rules.

There is still a number of questions that we have yet to answer, such as when is a function on either regular or nested datatypes a fold [GHA01], or an efficient fold? Moreover, if a function can be expressed as an efficient fold, under what conditions does this give a significant improvement in performance? On the theoretical side, there are similarities between the initial algebra definitions of generalised and efficient folds and those of Mendler-style inductive types [UV99]. Closer inspection has shown that our definitions do not fit into the existing framework of [UV99], but it might be possible to extend the theory to include them. There seems little point in pursuing this theory yet though, until we have found some more interesting applications of efficient folds and nested datatypes.

10. Acknowledgements

The authors would like to thank the anonymous referees, whose thorough reviewing significantly improved the presentation of this paper.

References

[BM98] R. Bird and L. Meertens. Nested datatypes. In *LNCS 1422: Mathematics of Program Construction*, p. 52–67. Springer, 1998.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[BP99a] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77-91, January 1999.

[BP99b] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:200-222, 1999.

[GHA01] J. Gibbons, G. Hutton and T. Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science* 44 No 1 (2001).

[Hin00] R. Hinze. Efficient generalized folds. In *Johan Jeuring, editor, Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal*, 6th July 2000.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calculus, *Theoretical Computer Science*, 1:27–57 (1975).

[Jon95] M.P. Jones. Functional programming with overloading and higher-order polymorphism. In *LNCS 925: First International Spring School on Advanced Functional Programming Techniques*, p. 97–136, Springer Verlag, 1995.

[Mac98] S. Mac Lane. *Categories for the Working Mathematician.* Second Edition. Springer-Verlag, 1998.

[MG01] C.E. Martin and J. Gibbons. On the semantics of nested datatypes. *Information Processing Letters*, 80:233-238 (2001).

[MA86] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Mee87] L. G. L. T. Meertens. *First steps towards the theory of rose trees.* Draft Report, CWI, Amsterdam, Netherlands, 1987.

[Oka98] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[PJ03] S. Peyton Jones (ed). *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press 2003.

[UV99] T. Uustalu, V. Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343-361, 1999.