Formal Aspects of Computing (1999) 3: 1–000

Generalised folds for nested datatypes

Richard Bird[1] and Ross Paterson[2]

Abstract. Nested datatypes generalise regular datatypes in much the same way that context-free languages generalise regular ones. Although the categorical semantics of nested types turns out to be similar to the regular case, the fold functions are more limited because they can only describe natural transformations. Practical considerations therefore dictate the introduction of a generalised fold function in which this limitation can be overcome. In the paper we show how to construct generalised folds systematically for each nested datatype, and show that they possess a uniqueness property analogous to that of ordinary folds. As a consequence, generalised folds satisfy fusion properties similar to those developed for regular datatypes. Such properties form the core of an effective calculational theory of inductive datatypes.

Keywords: Functional programming, program construction.

1. Introduction

A *nested* datatype is a parametrised inductive datatype whose declaration involves a change to the accompanying type parameter(s). In other words, the recursion is "nested" within a change of parameter. A formal definition is given in Section 3. Such types have also been called *non-regular* or *non-uniform*.

A simple example is provided by the type *Pow* of *power* trees, declared in Haskell by:

$$\begin{array}{lcl} \textbf{data}\ Pow\ a & = & Zero\ a \mid Succ\ (Pow\ (Pair\ a)) \\ \textbf{type}\ Pair\ a & = & a \times a \end{array}$$

(We have varied the syntax slightly, writing $a \times a$ where Haskell would use (a, a).)

Elements of $Pow\ a$ consist of pairs of pairs of pairs ... of values of type

[1] Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.
[2] Department of Computer Science, City University, Northhampton Square, London EC1V 0HB, UK.

a, where the depth of nesting is given by reading the associated constructor expression as a Peano numeral. For example,

$$Succ\,(Succ\,(Zero\,((1,2),(3,4))))$$

is a power tree of height 2. Each element therefore represents a perfectly balanced binary tree with labels in the leaves.

As this example suggests, nested datatypes can capture certain structural invariants in a way that regular datatypes cannot. Connelly and Lockwood Morris [CL95] use a nested type for modelling a generalisation of the trie data structure, and Okasaki [Oka98] puts nested datatypes to a variety of uses in the design of purely functional data structures. Bird and Paterson [BP99] use a nested datatype to describe de Bruijn notation for lambda expressions. Other examples of datatypes that have been shown very recently to be expressible by nested declarations include AVL trees, 2-3 trees, and square matrices.

The aim of this paper is to lay the groundwork for a useful calculational theory of nested datatypes; the work can be regarded as a successor to [BM98]. The categorical semantics is straightforward, but there is a complication. For a regular datatype, the fold function induced by initiality is a useful tool in the construction of practical programs. In nested datatypes, it isn't. The basic reason is that the argument of the fold is required to be a natural transformation, and the result of the fold is also a natural transformation. Consequently, functions that inspect the elements of a nested datatype cannot be defined as folds. This phenomenon motivates the introduction of a generalised fold operator in which these limitations can be overcome. The first contribution of the paper is to show how to define a generalised fold operator for each nested type.

In the calculational theory of regular inductive datatypes, everything hinges on the uniqueness property of the fold function: the fold is the unique function of its type satisfying a certain functional equation. From this flow the fusion laws of fold; such laws are of fundamental importance in reasoning about programs and improving their efficiency (see, e.g. [Bir98, Mal90b]). The second contribution of the paper is to show that generalised folds also possess a uniqueness property.

Having established the uniqueness property, we can set up appropriate fusion laws for reasoning about nested datatypes. The third contribution in this paper is to present such laws, and to discuss some of their consequences.

The rest of the paper is organised as follows. In Sections 2 and 3 we briefly review the standard theory of regular datatypes, and show how the theory extends to nested datatypes. These sections also contain the formal definitions of regular and nested datatypes. In Section 4 we introduce the generalised folds, and in Section 5 we present the fusion laws. In Section 6 we will justify our proofs of the fusion laws by showing that generalised folds are uniquely determined by their defining equations. Familiarity with basic category theory is assumed.

2. Semantics of non-parametrised datatypes

In a categorical setting, non-parametrised inductive datatypes are introduced by taking categorical least fixed points of appropriate functors (see e.g. [Lam70, BdM97, Hag87, Mal90a, MA86]). Given a suitable category $\mathbf{C}$, the least fixed point of a functor $F :: \mathbf{C} \to \mathbf{C}$, if it exists, is an object T of $\mathbf{C}$ for which the arrow $\alpha :: F\,T \to T$ is an initial algebra in the category of F-algebras. The objects of this category are arrows $F\,A \to A$ of $\mathbf{C}$, and the arrows are homomorphisms between algebras. Thus, for objects $f :: F\,A \to A$ and $g :: F\,B \to B$, an arrow $h :: f \to g$ is an arrow $h :: A \to B$ of $\mathbf{C}$ such that $h \cdot f = g \cdot F\,h$.

The initiality of α means that to each algebra $f :: F\,A \to A$ there is associated a unique homomorphism $h :: \alpha \to f$, that is a function $h :: T \to A$ satisfying the equation

$$h \cdot \alpha \quad = \quad f \cdot F\,h.$$

The unique homomorphism h is denoted by $\mathit{fold}\,f$. It is a consequence of initiality that α is an isomorphism, whence $T \cong F\,T$; this explains the terminology 'fixed point'.

One class of appropriate functors are the *polynomial* functors. In a category $\mathbf{C}$ with products and coproducts, the polynomial functors $F :: \mathbf{C}^n \to \mathbf{C}$ are defined by the following grammar:

$$F \quad ::= \quad \underline{A}^n \mid + \mid \times \mid \Pi_i^n \mid F \cdot \langle F_1, \ldots, F_n \rangle$$

The functor $\underline{A}^n$ is an n-ary constant functor, delivering the type A for all arguments. The functor Π_i^n is an n-ary projection, selecting its ith argument. A special case is $\mathit{Id} = \Pi_1^1$. The superscripts n will be omitted in the rest of the paper. The functor $F \cdot \langle F_1, \ldots, F_n \rangle$ denotes the composition of an n-ary functor F with n functors F_i, all of the same arity, so that

$$(F \cdot \langle F_1, \ldots, F_n \rangle)\, A_1 \ldots A_m \quad = \quad F\,(F_1\,A_1 \ldots A_m) \ldots (F_n\,A_1 \ldots A_m)$$

We omit the brackets when $n = 1$. We also write $F + G$ for $+ \cdot \langle F, G \rangle$ and similarly $F \times G$.

Provided the category $\mathbf{C}$ has colimits of all countable chains (the category $\mathbf{Set}$ of sets and total functions is an example), every unary polynomial functor possesses a least fixed point.

As examples, we can introduce the datatype Nat of natural numbers as the least fixed point of $F = \underline{1} + \mathit{Id}$, lists of natural numbers as the least fixed point of $F = \underline{1} + \underline{\mathit{Nat}} \times \mathit{Id}$, and binary trees, with natural number labels, as the least fixed point of $F = \underline{1} + \underline{\mathit{Nat}} \times \mathit{Id} \times \mathit{Id}$.

Such declarations can be translated into Haskell fairly directly. For example, the datatype of lists of natural numbers can be declared by

$$\begin{array}{lcl} \mathbf{data}\ \mathit{NatListF}\ a & = & \mathit{Nil} \mid \mathit{Cons}\,(\mathit{Nat} \times a) \\ \mathbf{newtype}\ \mathit{NatList} & = & \mathit{In}\,(\mathit{NatListF}\ \mathit{NatList}) \end{array}$$

The **data** declaration introduces the polynomial functor *NatListF*, and the **newtype** declaration introduces *NatList* as isomorphic to *NatListF NatList*. In Haskell, both type variables and type constructor variables are denoted using lower-case letters.

In general, Haskell programs may fail to terminate, so these datatypes conceptually include an additional 'value' representing non-termination, and the language defines how non-termination propagates through an expression. However, we shall be using a subset of Haskell in which all programs terminate, so that this is not an issue, and our programs may be modelled in our category $\mathbf{C}$. This means that we cannot use unrestricted recursion; we must show that any recursive definitions we use have unique solutions. An additional restriction in comparison with Haskell is that recursively defined types in this paper do not involve function types, so we are concerned solely with inductive types.

The constructor *In* corresponds to the initial algebra α. The converse function α^{-1} is given by the function *out*, defined in Haskell by

$$\begin{array}{lcl} \mathit{out} & :: & \mathit{NatList} \to \mathit{NatListF}\ \mathit{NatList} \\ \mathit{out}\,(\mathit{In}\,x) & = & x \end{array}$$

The type of lists of natural numbers can also be declared by

$$\mathbf{data}\ \mathit{NatList} \quad = \quad \mathit{Nil} \mid \mathit{Cons}\,(\mathit{Nat} \times \mathit{NatList})$$

In this version, which is closer to the usual style of datatype declaration in functional programming, the polynomial functor *NatListF* does not appear explicitly.

The function *fold* may be defined in Haskell as follows:

$$\begin{array}{lcl} \mathit{fold} & :: & (\mathit{NatListF}\ a \to a) \to \mathit{NatList} \to a \\ \mathit{fold}\,f & = & f \cdot \mathit{natlistF}\,(\mathit{fold}\,f) \cdot \mathit{out} \end{array}$$

In Haskell, a free type variable, like a, is implicitly universally quantified, and may be instantiated to any type.

The function *natlistF* is the functorial action of *NatListF*:

$$\begin{array}{lcl} \mathit{natlistF} & :: & (a \to b) \to \mathit{NatListF}\ a \to \mathit{NatListF}\ b \\ \mathit{natlistF}\,f\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{natlistF}\,f\,(\mathit{Cons}\,(n, x)) & = & \mathit{Cons}\,(n, f\,x) \end{array}$$

For example, $\mathit{sumNL} = \mathit{fold}\ \mathit{sumF}$ sums a list, where

$$\begin{array}{lcl} \mathit{sumF} & :: & \mathit{NatListF}\ \mathit{Nat} \to \mathit{Nat} \\ \mathit{sumF}\ \mathit{Nil} & = & 0 \\ \mathit{sumF}\,(\mathit{Cons}\,(m, n)) & = & m + n \end{array}$$

3. Semantics of parametrised datatypes

Next, we consider the extension of this theory to parametrised datatypes, like the following Haskell definition for lists:

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

As before, this may be rewritten in the equivalent form

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Base}\ a\ b & = & \mathit{Nil} \mid \mathit{Cons}\,(a \times b) \\ \mathbf{newtype}\ \mathit{List}\ a & = & \mathit{In}\,(\mathit{Base}\ a\,(\mathit{List}\ a)) \end{array}$$

To incorporate parametrised datatypes into the theory, one can proceed in essentially two different ways. The first method is standard, but the second extends to nested constructors. For simplicity, we will consider only datatypes parametrised by a single type variable; the extension to multiple parameters is straightforward.

3.1. First method: families of fixed points

The first method takes the fixed point of a binary functor B with respect to its second parameter. We begin with the observation that the partial application of a binary functor B to a type A yields a unary functor $B\,A$, whose least fixed point $T\,A$ can be constructed in the usual way. Then for each A there is an associated initial algebra α_A of type

$$\alpha_A \quad :: \quad B\,A\,(T\,A) \to T\,A.$$

The associated fold function takes an algebra $f :: B\ A\ Y \to Y$ as argument, and is uniquely defined by the assertion that for any arrow $h :: T\ A \to Y$, we have $h = \mathit{fold}\, f$ if and only if

$$h \cdot \alpha_A \quad = \quad f \cdot B\ \mathit{id}\ h.$$

The datatype constructor T can be made into a functor by defining $T\, f :: T\ X \to T\ Y$ for an arrow $f :: X \to Y$ by

$$T\, f \quad = \quad \mathit{fold}\,(\alpha_Y \cdot B\, f\ \mathit{id}).$$

Restated, this definition takes the form

$$T\, f \cdot \alpha_X \quad = \quad \alpha_Y \cdot B\, f\,(T\, f).$$

Hence α is a natural transformation from $B \cdot \langle \mathit{Id}, T\rangle$ to T.

The class of *regular* functors is defined as the closure of the class of polynomial bifunctors under least fixed point operations. For example, *List* is the least fixed point of a bifunctor $B = \underline{1} + \Pi_1 \times \Pi_2$, and the general tree constructor *Tree* is the least fixed point of a bifunctor $B = \Pi_1 \times (\mathit{List} \cdot \Pi_2)$.

For the *List* datatype constructor, the fold function is implemented by

$$\begin{array}{lcl} \mathit{fold} & :: & (\mathit{Base}\ a\ b \to b) \to \mathit{List}\ a \to b \\ \mathit{fold}\, f & = & f \cdot \mathit{base}\ \mathit{id}\,(\mathit{fold}\, f) \cdot \mathit{out} \end{array}$$

where the function *base* implements the functorial action of *Base*:

$$\begin{array}{lcl} \mathit{base} & :: & (a \to c) \to (b \to d) \to \mathit{Base}\ a\ b \to \mathit{Base}\ c\ d \\ \mathit{base}\, f\ g\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{base}\, f\ g\,(\mathit{Cons}\,(x, y)) & = & \mathit{Cons}\,(f\ x, g\ y) \end{array}$$

Though it is defined in a slightly different way than usual, the function *fold* is essentially the same as the standard function *foldr* in functional programming. The functorial action of *List* (corresponding to the standard function *map*) may be defined using *fold*:

$$\begin{array}{lcl} \mathit{list} & :: & (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b \\ \mathit{list}\, f & = & \mathit{fold}\,(\mathit{In} \cdot \mathit{base}\, f\ \mathit{id}) \end{array}$$

3.2. Second method: higher-order fixed points

The second method yields a larger class of datatype constructors. Instead of constructing fixed points in the category $\mathbf{C}$, we will work in the functor categories $\mathbf{C}^n \to \mathbf{C}$. The objects of $\mathbf{C}^n \to \mathbf{C}$ are n-ary functors over $\mathbf{C}$, and the arrows are natural transformations between functors. We shall write $F \to G$ for the set of natural transformations from F to G.

In this category, limits and colimits may be constructed pointwise from limits and colimits in $\mathbf{C}$ [Mac71, V.3]. For example, the shorthand notations $F \times G$ and $F + G$ do in fact denote products and coproducts in this category[3].

Moreover, we can define a functor $T :: \mathbf{C}^n \to \mathbf{C}$ as a fixed point of a suitable higher-order functor $F :: (\mathbf{C}^n \to \mathbf{C}) \to (\mathbf{C}^n \to \mathbf{C})$, via the same colimit construction used in $\mathbf{C}$. We shall refer to such functors as *hofunctors*. As before,

[3] Everything we do here generalises in a straightforward way to functor categories $\mathbf{D} \to \mathbf{C}$, for an arbitrary category $\mathbf{D}$.

this means that $\alpha :: F\ T \to T$ is an initial object in the category of F-algebras. Note that α is a natural transformation by construction. Hence, for $f :: A \to B$, we have

$$T\,f \cdot \alpha_A \quad = \quad \alpha_B \cdot F\ T\,f.$$

The fold function induced by the initiality of α associates with each natural transformation $f :: F\ N \to N$, for some given functor N, a natural transformation $\mathit{hfold}\,f :: T \to N$ defined as the unique function h satisfying

$$h \cdot \alpha \quad = \quad f \cdot F\,h.$$

This is exactly the same definition as for *fold* on unparametrised datatypes, but installed at a higher level. One major difference though is that the functoriality of T is given by construction, not as an instance of the fold.

We define the class of *polynomial hofunctors* of the form $F\,X = P$, where P is a polynomial functor expression that may also include the functor variable X. The grammar for such functor expressions is

$$P \quad ::= \quad \underline{A}^n \mid + \mid \times \mid \Pi^n_i \mid P \cdot \langle P_1, \ldots, P_n \rangle \mid X$$

A fixed point of a polynomial hofunctor is called a *nested* functor[4]. We consider three examples.

Example 3.1 The functor *List* is given as the least fixed point of the polynomial hofunctor $\mathit{ListF}\ X = \underline{1} + \mathit{Id} \times X$. Thus *List* is a nested functor which is also a regular functor. In Haskell, the type constructor *List* may be introduced by the declarations

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Base}\ a\ b & = & \mathit{Nil} \mid \mathit{Cons}\ (a \times b) \\
\textbf{type}\ \mathit{ListF}\ x\ a & = & \mathit{Base}\ a\ (x\ a) \\
\textbf{newtype}\ \mathit{List}\ a & = & \mathit{In}\ (\mathit{ListF}\ \mathit{List}\ a)
\end{array}$$

The **type** declaration introduces $\mathit{ListF}\ x\ a$ as a synonym for $\mathit{Base}\ a\ (x\ a)$. The type constructor *List* is therefore exactly the same as before.

However, the fold function is different. This function may be expressed in Haskell as follows:

$$\begin{array}{lcl}
\mathit{hfold} & :: & (\forall\, a.\, \mathit{ListF}\ n\ a \to n\ a) \to \mathit{List}\ b \to n\ b \\
\mathit{hfold}\ f & = & f \cdot \mathit{listF}\ (\mathit{hfold}\ f) \cdot \mathit{out}
\end{array}$$

The function *listF* expresses part of the functorial action of *ListF*: for each natural transformation $f :: X \to Y$, we have $\mathit{listF}\ f :: \mathit{ListF}\ X \to \mathit{ListF}\ Y$. In Haskell, we have

$$\begin{array}{lcl}
\mathit{listF} & :: & (\forall\, a.\, x\ a \to y\ a) \to \mathit{ListF}\ x\ b \to \mathit{ListF}\ y\ b \\
\mathit{listF}\ f & = & \mathit{base}\ \mathit{id}\ f
\end{array}$$

The first arguments of *hfold* and *listF* are polymorphic functions, described using a *rank 2* type signature [McC84]. Such signatures are allowed in recent extensions to Haskell [Jon98, PL97]. Polymorphic functions correspond to natural transformations in a categorical setting.

The definition of *hfold* is equivalent to the earlier definition of the regular *fold*, but has a different type: the new version takes natural transformations to

[4] Though we deal only with single recursive definitions here, our results could be extended to systems of multiple simultaneous definitions.

natural transformations. As a result, the new fold is less useful because individual list elements can never be inspected.

For example, if a fold is to produce a natural number, we would have to take $N = \underline{Nat}$, the constant functor that returns Nat on every type, so that $hfold\, f :: List \to \underline{Nat}$. This function will operate on lists of any type, and must satisfy the naturality property

$$hfold\, f \quad = \quad hfold\, f \cdot list\, k$$

for any function k, and thus cannot depend on elements of the list. We can define the length function, but little more. In the following section we will see how to overcome this limitation.

We have noted above that a functor $List$ is introduced automatically as part of the categorical semantics, but we have not yet defined its functorial action $list$ in Haskell. An inductive argument establishes that $list$ is uniquely determined by the naturality of $In :: ListF\ List \to List$. To express this naturality condition, we could extend the definition of $listF$ given above to the full functorial action of $ListF$, or simply expand $ListF$ in the definition of $List$, yielding the definition

$$\begin{array}{lcl} list & :: & (a \to b) \to List\ a \to List\ b \\ list\, f & = & In \cdot base\, f\ (list\, f) \cdot out \end{array}$$

This is equivalent to the definition using $fold$ at the end of the previous section. Note, however, that $list$ cannot be defined using $hfold$.

Example 3.2 The type constructor $Nest$ is defined as the least fixed point of

$$NestF\ X \quad = \quad \underline{1} + Id \times (X \cdot Pair).$$

In Haskell the corresponding declarations are:

$$\begin{array}{lcl} \textbf{data}\ Base\ a\ b & = & Nil \mid Cons\,(a \times b) \\ \textbf{type}\ NestF\ x\ a & = & Base\ a\,(x\,(Pair\ a)) \\ \textbf{type}\ Pair\ a & = & a \times a \\ \textbf{newtype}\ Nest\ a & = & In\,(NestF\ Nest\ a) \end{array}$$

An equivalent datatype is introduced by the ‘flattened’ declaration

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

Unlike $List$, the datatype constructor $Nest$ is not a regular functor: occurrences of $Nest$ in its defining expression are “nested” within a change of type parameter from a to $Pair\ a$. In effect, values of type $Nest\ a$ correspond to lists in which the first element has type a, the second element has type $(a \times a)$, the third has type $((a \times a) \times (a \times a))$, and so on. A variant of $Nest$ was used in [Oka98] as a basis for an implementation of lists with an efficient indexing operation.

The fold function for $Nest$ may be implemented in Haskell by expanding $NestF$ in terms of the bifunctor $Base$:

$$\begin{array}{lcl} hfold & :: & (\forall\, a.\, Base\ a\,(n\,(Pair\ a)) \to n\ a) \to Nest\ b \to n\ b \\ hfold\, f & = & f \cdot base\ id\,(hfold\, f) \cdot out \end{array}$$

Similarly, the mapping function $nest$ may be defined by

$$\begin{array}{lcl} \mathit{nest} & :: & (a \to b) \to \mathit{Nest}\ a \to \mathit{Nest}\ b \\ \mathit{nest}\, f & = & \mathit{In} \cdot \mathit{base}\, f\ (\mathit{nest}\ (\mathit{pair}\, f)) \cdot \mathit{out} \end{array}$$

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

Unlike the case of List, the naturality of the argument f in $\mathit{hfold}\, f$ is crucial to the definition. The recursive occurrence of $\mathit{hfold}\, f$ in its defining expression is applied to elements of type $\mathit{Nest}\ (\mathit{Pair}\ a)$; consequently, f must have type

$$f \quad :: \quad \mathit{Base}\ (\mathit{Pair}^k\ b)\ (n\ (\mathit{Pair}^{k+1}\ b)) \to n\ (\mathit{Pair}^k\ b)$$

for all $k \geq 0$. The type signature of f in the definition of hfold generalises this requirement by making f fully polymorphic.

Note that in the definition of Nest, the parameters of the recursive use do not themselves contain a use of Nest. We call such nested datatypes *linear*; almost all of the known practical examples of nested datatypes belong to this class. For an exception, see Section 5 of [BP99]. Our final example is a non-linear variant of Nest.

Example 3.3 The type functor Host may be defined as the least fixed point of

$$\mathit{HostF}\ X \quad = \quad \underline{1} + \mathit{Id} \times (X \cdot (\mathit{Id} \times X)),$$

The Haskell declaration is:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Base}\ a\ b & = & \mathit{Nil} \mid \mathit{Cons}\ (a \times b) \\ \textbf{type}\ \mathit{HostF}\ x\ a & = & \mathit{Base}\ a\ (x\ (a \times x\ a)) \\ \textbf{newtype}\ \mathit{Host}\ a & = & \mathit{In}\ (\mathit{HostF}\ \mathit{Host}\ a) \end{array}$$

This datatype is equivalent to the flattened version

$$\textbf{data}\ \mathit{Host}\ a \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ (a \times \mathit{Host}\ (a \times \mathit{Host}\ a))$$

In effect, values of type $\mathit{Host}\ a$ correspond to lists in which the first element is of type a, the second element is of type $a \times \mathit{Host}\ a$, the third element is of type

$$(a \times \mathit{Host}\ a) \times \mathit{Host}\ (a \times \mathit{Host}\ a),$$

and so on. We know of no practical use for this datatype.

The fold and map functions may be implemented by

$$\begin{array}{lcl} \mathit{hfold} & :: & (\forall\, a.\, \mathit{Base}\ a\ (n\ (a \times n\ a)) \to n\ a) \to \mathit{Host}\ b \to n\ b \\ \mathit{hfold}\, f & = & f \cdot \mathit{base}\ \mathit{id}\ (\mathit{hfold}\, f \cdot \mathit{host}\ (\mathit{id} \times \mathit{hfold}\, f)) \cdot \mathit{out} \end{array}$$

$$\begin{array}{lcl} \mathit{host} & :: & (a \to b) \to \mathit{Host}\ a \to \mathit{Host}\ b \\ \mathit{host}\, f & = & \mathit{In} \cdot \mathit{base}\, f\ (\mathit{host}\ (f \times \mathit{host}\, f)) \cdot \mathit{out} \end{array}$$

$$\begin{array}{lcl} \times & :: & (a \to c) \to (b \to d) \to a \times b \to c \times d \\ (f \times g)\ (x, y) & = & (f\ x, g\ y) \end{array}$$

The functions host and $\times$ implement the functorial actions of Host and $\times$, respectively.

4. Generalised folds

A simple fold takes as argument a natural transformation $f :: F\,N \to N$ and returns a natural transformation of type $T \to N$. Its definition has the form

$$hfold\,f \cdot \alpha \quad = \quad \Psi\,(hfold\,f)$$

where the form $\Psi :: (T \to N) \to F\;T \to N$ is defined by $\Psi\,h = f \cdot F\,h$.

Greater flexibility is achieved if we generalise the fold to return a natural transformation of type $T \cdot M \to N$. For example, both M and N could be taken to be the constant functor that returns the type of integers. Then we would have a way of reducing a T-structure of integers to a single integer. The generalised fold will take as arguments a series of natural transformations $f, g_1, \ldots, g_m$, to be introduced below. A generalised fold

$$gfold\,f\;g_1\;\cdots\;g_m :: T \cdot M \to N$$

will be defined by an equation of the form

$$gfold\,f\;g_1\;\cdots\;g_m \cdot \alpha \quad = \quad \Psi\,(gfold\,f\;g_1\;\cdots\;g_m)$$

for some form $\Psi :: (T \cdot M \to N) \to F\;T \cdot M \to N$. To develop this operator Ψ, observe that every polynomial hofunctor F can be expressed in the form

$$F\,X \quad = \quad B \cdot \langle Id, X \cdot F_1\,X, \ldots, X \cdot F_n\,X \rangle$$

for some $n \geq 0$ (if $n = 0$, then $F\,X = B \cdot Id = B$). The subsidiary hofunctors F_i may be decomposed similarly[5].

For example, in the case of *List*, *Nest*, or *Host*, the functor B is given by

$$B\,X\;Y \quad = \quad 1 + X \times Y.$$

Now for $h :: T \cdot M \to N$, we require (expanding $F\;T \cdot M$):

$$\Psi\,h \quad :: \quad B \cdot \langle M, T \cdot F_1\;T \cdot M, \ldots, T \cdot F_n\;T \cdot M \rangle \to N.$$

The component types $T \cdot F_i\;T \cdot M$ are not suitable for arguments to h, so we introduce auxiliary forms $\Phi[F_i]\,h :: F_i\;T \cdot M \to M \cdot F_i\,N$, to be defined below. Then we have

$$h \cdot T\,(\Phi[F_i]\,h) \quad :: \quad T \cdot F_i\;T \cdot M \to N \cdot F_i\,N$$

and we can define

$$\Psi\,h \quad = \quad f \cdot B\,id\,(h \cdot T\,(\Phi[F_1]\,h))\;\cdots\;(h \cdot T\,(\Phi[F_n]\,h))$$

for some natural transformation

$$f \quad :: \quad B \cdot \langle M, N \cdot F_1\,N, \ldots, N \cdot F_n\,N \rangle \to N$$

which must be supplied as a parameter to *gfold*.

It remains to define $\Phi[F]h$, which we do inductively on the form of F. For $F\,X = B \cdot \langle Id, X \cdot F_1\,X, \ldots, X \cdot F_n\,X \rangle$, we define

$$\Phi[F]h \quad = \quad g \cdot B\,id\,(h \cdot T\,(\Phi[F_1]h))\;\cdots\;(h \cdot T\,(\Phi[F_n]h))$$

for some natural transformation

$$g \quad :: \quad B \cdot \langle M, N \cdot F_1\,N, \ldots, N \cdot F_n\,N \rangle \to M \cdot F\,N.$$

[5] There may be several such decompositions, each giving rise to a different definition of *gfold*.

Each natural transformation g arising in this way is provided as an extra parameter to *gfold*.

Note that in the special case $M = Id$, we can set each $g = id$, obtaining the simple fold as a special case:

$$hfold\,f \quad = \quad gfold\,f\;id\;\cdots\;id.$$

Another special case arises when both M and N are constant functors; the associated instance of *gfold* is called a *reduction*.

Let us now consider the detailed instantiations for each of our three example datatypes.

Example 4.1 For *List* we have $List = F\;List$, where

$$\begin{aligned} F\,X &= Base \cdot \langle Id, X \cdot F_1\,X \rangle \\ F_1\,X &= Id. \end{aligned}$$

The generalised fold therefore takes the following form in Haskell:

$$\begin{aligned} gfold \quad &:: \quad (\forall\,a.\,Base\,(m\;a)\,(n\;a) \to n\;a) \to \\ &\qquad (\forall\,a.\,m\;a \to m\;a) \to \\ &\qquad List\,(m\;b) \to n\;b \\ gfold\,f\;g \quad &= \quad f \cdot base\;id\,(gfold\,f\;g \cdot list\;g) \cdot out \end{aligned}$$

Of the two additional ingredients, M and g, it is the presence of the functor M that is the crucial one; the second argument g is useful but its effect can be achieved by other means. The point about M is that it provides greater freedom of manoeuvre.

As noted above, we can take both M and N to be constant functors, delivering, say, the types x and y respectively. Then the associated instance of *gfold* is a reduction of type

$$reduce \quad :: \quad (Base\,x\,y \to y) \to (x \to x) \to List\,x \to y.$$

Viewing $f :: Base\,x\,y \to y$ as a pair $e :: y$ and $\oplus :: x \times y \to y$, the effect of $reduce\,f\;g$ on a list $[a_0, \ldots, a_n]$ is to produce the value

$$a_0 \oplus (g\;a_1 \oplus (g^2\;a_2 \oplus \cdots (g^n\;a_n \oplus e) \cdots)).$$

The regular *fold* is the special case of a list reduction in which $g = id$. List reductions are an important component of the circuit design language Ruby [JS93]; see also [BdM97].

This analysis of reductions suggests that a generalised fold over lists can be decomposed into a simple fold after a *triangle*. The triangle with respect to a function g takes $[a_0, a_1, \ldots, a_n]$ to $[a_0, g\;a_1, \ldots, g^n\;a_n]$. This function can be expressed as a fold:

$$\begin{aligned} triangle \quad &:: \quad (a \to a) \to List\;a \to List\;a \\ triangle\;g \quad &= \quad fold\,(In \cdot base\;id\,(list\;g)). \end{aligned}$$

The claim is that

$$gfold\,f\;g \quad = \quad fold\,f \cdot triangle\;g.$$

This result will be proved in the following section. In fact, for regular datatypes, we can always decompose a generalised fold into the composition of two regular folds in this way.

Example 4.2 For *Nest* we have $Nest = F\,Nest$, where

$$\begin{array}{rcl} F\,X & = & Base \cdot \langle Id, X \cdot F_1\,X \rangle \\ F_1\,X & = & Pair. \end{array}$$

The generalised fold therefore has the following form in Haskell:

$$\begin{array}{lcl} gfold & :: & (\forall\, a.\, Base\,(m\,a)\,(n\,(Pair\,a)) \rightarrow n\,a) \rightarrow \\ & & (\forall\, a.\, Pair\,(m\,a) \rightarrow m\,(Pair\,a)) \rightarrow \\ & & Nest\,(m\,b) \rightarrow n\,b \\ gfold\,f\,g & = & f \cdot base\,id\,(gfold\,f\,g \cdot nest\,g) \cdot out \end{array}$$

If we take M and N to be constant functors, delivering types x and y respectively, then we obtain an instance of *gfold*, again called a reduction, with type:

$$reduce :: (Base\,x\,y \rightarrow y) \rightarrow (Pair\,x \rightarrow x) \rightarrow Nest\,x \rightarrow y$$

In particular, we can sum a nest of integers by $sumN = reduce\,sumB\,plus$, where $plus\,(m, n) = m + n$ and

$$\begin{array}{lcl} sumB & :: & Base\,Int\,Int \rightarrow Int \\ sumB\,Nil & = & 0 \\ sumB\,(Cons\,(m, n)) & = & m + n \end{array}$$

A nest of integers cannot be summed using a fold over nests (at least, not simply) because *sumB* is not a natural transformation of the right type. The summation function can be defined using a generalised fold because every function between two given types lifts to a natural transformation between constant functors.

Example 4.3 For *Host* we have $Host = F\,Host$, where

$$\begin{array}{rcl} F\,X & = & Base \cdot \langle Id, X \cdot F_1\,X \rangle \\ F_1\,X & = & \times \cdot \langle Id, X \cdot F_2\,X \rangle \\ F_2\,X & = & Id. \end{array}$$

The generalised fold therefore has the following form in Haskell:

$$\begin{array}{lcl} gfold & :: & (\forall\, a.\, Base\,(m\,a)\,(n\,(a \times n\,a)) \rightarrow n\,a) \rightarrow \\ & & (\forall\, a.\, m\,a \times n\,a \rightarrow m\,(a \times n\,a)) \rightarrow \\ & & (\forall\, a.\, m\,a \rightarrow m\,a) \rightarrow \\ & & Host\,(m\,b) \rightarrow n\,b \\ gfold\,f\,g_1\,g_2 & = & f \cdot base\,id\,(gfold\,f\,g_1\,g_2\,\cdot \\ & & \qquad host\,(g_1 \cdot id \times (gfold\,f\,g_1\,g_2 \cdot host\,g_2))) \cdot out \end{array}$$

Taking M and N to be constant functors, delivering types x and y respectively, we can define a reduction over *Host* as an instance of *gfold* with type

$$reduce \;::\; (Base\,x\,y \rightarrow y) \rightarrow (x \times y \rightarrow x) \rightarrow (x \rightarrow x) \rightarrow Host\,x \rightarrow y$$

In particular, we can sum a host of integers by $sumH = reduce\,sumB\,plus\,id$, using the functions *sumB* and *plus* defined above.

4.1. Remarks

As we have defined them, generalised folds can be criticised from two opposing points of view. One can argue that our folds are *too* general in that arguments of type $M \to M$ could be dropped, since it is not clear what they contribute. On the other hand, one can also argue that our folds are too *specific*. For example, it is quite possible to assign the following type signature to the above definition of *gfold* for *Host*:

$$\begin{array}{lll} \mathit{gfold} & :: & (\forall\, a.\, \mathit{Base}\,(m\; a)\,(n\,(p\; a)) \to n\; a) \to \\ & & (\forall\, a.\, m\; a \times n\,(q\; a) \to m\,(p\; a)) \to \\ & & (\forall\, a.\, m\; a \to m\,(q\; a)) \to \\ & & \mathit{Host}\,(m\; b) \to n\; b \end{array}$$

The idea is that $\mathit{gfold}\, f\; g\; h :: \mathit{Host}\,(m\; b) \to n\; b$ is a natural transformation, and different instances may be instantiated at different types, reflected by the type constructors q and p. The previous type of *gfold* is then the instance in which $q\; a = a$ and $p\; a = a \times n\; a$.

Even more generally, we can recall that the generalised fold is defined as the unique solution of an equation

$$x \cdot \alpha \quad = \quad \Psi\, x.$$

We can use any form $\Psi :: (T \cdot M \to N) \to (F\; T \cdot M \to N)$ for which this equation has a unique solution. As we shall see in Section 6, it suffices that Ψ is defined in such a way that any functor may be used in place of T. In particular, Ψ may not use α or α^{-1} to build or dismantle values of type T. We write

$$\Psi \quad :: \quad \forall\, X.\, (X \cdot M \to N) \to (F\; X \cdot M \to N).$$

Thus Ψ is a natural transformation between two contravariant functors.

As always, the more general forms have more degrees of freedom, but consequently offer less guidance to the program designer. We have found the version described in this section, with associated fusion laws presented below, to be a reasonable compromise in practice [BP99]. However, more experience is needed to determine which of the formulations turn out to be the most useful in practical program construction.

5. Fusion laws

The fusion laws for a nested datatype T come in two flavours: *fold-fusion* and *map-fusion*. The former provides conditions under which

$$k \cdot \mathit{gfold}\, f\; g_1 \cdots g_m \quad = \quad \mathit{gfold}\, f'\; g_1' \cdots g_m'.$$

The map-fusion law is similar, providing conditions under which

$$\mathit{gfold}\, f\; g_1 \cdots g_m \cdot T\; k \quad = \quad \mathit{gfold}\, f'\; g_1' \cdots g_m'.$$

In the standard theory of regular datatypes, map-fusion is a special case of fold-fusion because the functorial action of regular datatypes can be defined as a fold. This is not the case with the higher-order semantics of Section 3.

In deriving the fusion laws, we shall assume that $\mathit{gfold}\, f'\; g_1' \cdots g_m'$ is the unique function h' satisfying its defining equation. This assumption will be justified in Section 6.

As in the previous section, we suppose $\alpha :: F\; T \to T$, where

$$F\, X \quad = \quad B \cdot \langle \mathit{Id}, X \cdot F_1\, X, \ldots, X \cdot F_n\, X \rangle.$$

To reduce clutter, we show the calculations only for the case $n = 1$; the generalisation is straightforward.

The map-fusion law is dealt with first.

5.1. Map-fusion laws

Abbreviating $gfold\, f\ g_1 \cdots g_m$ to h, and $gfold\, f'\ g'_1 \cdots g'_m$ to h', our aim is to give conditions under which $h \cdot T\, k = h'$, where $k :: M' \to M$. As noted above, we assume that h' is the unique function of type $T \cdot M' \to N$ satisfying

$$h' \cdot \alpha \quad = \quad \Psi'\, h',$$

where Ψ' is obtained by replacing each f and g_i in Ψ with the corresponding f' or g'_i. Hence it is sufficient to give conditions under which

$$h \cdot T\, k \cdot \alpha \quad = \quad \Psi'\, (h \cdot T\, k).$$

The general scheme is given by the following calculation:

$$\begin{array}{ll}
 & h \cdot T\, k \cdot \alpha \\
= & \quad \{\text{naturality of } \alpha\} \\
 & h \cdot \alpha \cdot B\, k\, (T\, (F_1\ T\, k)) \\
= & \quad \{\text{definition of } h\} \\
 & f \cdot B\, id\, (h \cdot T\, (\Phi[F_1]h)) \cdot B\, k\, (T\, (F_1\ T\, k)) \\
= & \quad \{\text{functoriality of } B\} \\
 & f \cdot B\, k\, (h \cdot T\, (\Phi[F_1]h) \cdot T\, (F_1\ T\, k)) \\
= & \quad \{\text{functoriality of } T\} \\
 & f \cdot B\, k\, (h \cdot T\, (\Phi[F_1]h \cdot F_1\ T\, k)) \\
= & \quad \{\textbf{assume } \Phi[F_1]h \cdot F_1\ T\, k = k \cdot \Phi'[F_1](h \cdot T\, k)\} \\
 & f \cdot B\, k\, (h \cdot T\, (k \cdot \Phi'[F_1](h \cdot T\, k))) \\
= & \quad \{\text{functoriality of } T\} \\
 & f \cdot B\, k\, (h \cdot T\, k \cdot T(\Phi'[F_1](h \cdot T\, k))) \\
= & \quad \{\text{functoriality of } B\} \\
 & f \cdot B\, k\, id \cdot B\, id\, (h \cdot T\, k \cdot T(\Phi'[F_1](h \cdot T\, k))) \\
= & \quad \{\textbf{assume } f' = f \cdot B\, k\, id\} \\
 & f' \cdot B\, id\, (h \cdot T\, k \cdot T(\Phi'[F_1](h \cdot T\, k))) \\
= & \quad \{\text{definition}\} \\
 & \Psi'\, (h \cdot T\, k).
\end{array}$$

The first assumption in this calculation is the condition

$$\Phi[F_1]h \cdot F_1\ T\, k \quad = \quad k \cdot \Phi'[F_1](h \cdot T\, k),$$

which can be simplified using the inductive definitions of Φ and Φ' with a similar calculation. Eventually we will arrive at subforms that are independent of h, having generated a series of side-conditions. We will spell out the details for our three example datatypes.

Example 5.1 For *List*, we have $\Phi[F_1]h = g :: M \to M$ and $F_1\, \mathit{List} = \mathit{Id}$. Since $\Phi'[F_1](h \cdot \mathit{list}\, k) = g' :: M' \to M'$, we obtain

MAP-FUSION LAW FOR *List*.

$$\mathit{gfold}\, f\; g \cdot \mathit{list}\, k = \mathit{gfold}\, (f \cdot \mathit{base}\, k\; \mathit{id})\; g' \quad \Leftarrow \quad g \cdot k = k \cdot g'.$$

Taking the special case $g = \mathit{id}$ and $g' = \mathit{id}$, we have

$$\mathit{fold}\, f \cdot \mathit{list}\, k \quad = \quad \mathit{fold}\, (f \cdot \mathit{base}\, k\; \mathit{id}).$$

Example 5.2 For *Nest*, we have $\Phi[F_1]h = g :: \mathit{Pair} \cdot M \to M \cdot \mathit{Pair}$ and $F_1\, \mathit{Nest} = \mathit{Pair}$. Since $\Phi'[F_1](h \cdot \mathit{nest}\, k) = g' :: \mathit{Pair} \cdot M' \to M' \cdot \mathit{Pair}$, we obtain

MAP-FUSION LAW FOR *Nest*.

$$\mathit{gfold}\, f\; g \cdot \mathit{nest}\, k = \mathit{gfold}\, (f \cdot \mathit{base}\, k\; \mathit{id})\; g' \quad \Leftarrow \quad g \cdot \mathit{pair}\, k = k \cdot g'.$$

Example 5.3 For *Host*, we have

$$\begin{aligned} \Phi[F_1]h \quad &= \quad g_1 \cdot \mathit{id} \times (h \cdot \mathit{host}\, g_2) \\ \Phi'[F_1](h \cdot \mathit{host}\, k) \quad &= \quad g_1' \cdot \mathit{id} \times (h \cdot \mathit{host}\, k \cdot \mathit{host}\, g_2'), \end{aligned}$$

where $g_1 :: M \times N \to M \cdot (\mathit{Id} \times N)$ and $g_2 :: M \to M$. The types of g_1' and g_2' are similar, except that M is replaced by M'. Since $F_1\, \mathit{Host}\, k = k \times \mathit{host}\, k$, the condition for map-fusion takes the form

$$g_1 \cdot \mathit{id} \times (h \cdot \mathit{host}\, g_2) \cdot k \times \mathit{host}\, k \quad = \quad k \cdot g_1' \cdot \mathit{id} \times (h \cdot \mathit{host}\, k \cdot \mathit{host}\, g_2')$$

or equivalently

$$g_1 \cdot k \times \mathit{id} \cdot \mathit{id} \times (h \cdot \mathit{host}\, (g_2 \cdot k)) \quad = \quad k \cdot g_1' \cdot \mathit{id} \times (h \cdot \mathit{host}\, (k \cdot g_2')).$$

This equation follows from $g_1 \cdot k \times \mathit{id} = k \cdot g_1'$ and $g_2 \cdot k = k \cdot g_2'$. Hence

MAP-FUSION LAW FOR *Host*.

$$\begin{aligned} &\mathit{gfold}\, f\; g_1\; g_2 \cdot \mathit{host}\, k = \mathit{gfold}\, (f \cdot \mathit{base}\, k\; \mathit{id})\; g_1'\; g_2' \\ &\qquad\qquad \Leftarrow \quad g_1 \cdot k \times \mathit{id} = k \cdot g_1' \;\wedge\; g_2 \cdot k = k \cdot g_2'. \end{aligned}$$

5.2. Fold-fusion laws

Again abbreviating $\mathit{gfold}\, f\; g_1 \cdots g_m$ to h and $\mathit{gfold}\, f'\; g_1' \cdots g_m'$ to h', our aim is to give conditions under which $k \cdot h_{M'} = h'$, where $k :: N \cdot M' \to N'$. As before, it is sufficient to give conditions under which

$$k \cdot h \cdot \alpha \quad = \quad \Psi'\, (k \cdot h).$$

The general scheme is given by the following calculation:

$$\begin{aligned} & k \cdot h \cdot \alpha \\ = \quad & \quad \{\text{definition of } h\} \\ & k \cdot f \cdot B\, \mathit{id}\, (h \cdot T(\Phi[F_1]\, h)) \\ = \quad & \quad \{\textbf{assume } k \cdot f = f' \cdot B\, \mathit{id}\, (k \cdot N\, p_1) \text{ for some } p_1\} \end{aligned}$$

$$
\begin{array}{ll}
& f' \cdot B\,id\,(k \cdot N\,p_1) \cdot B\,id\,(h \cdot T\,(\Phi[F_1]\,h)) \\
= & \quad \{\text{functoriality of } B\} \\
& f' \cdot B\,id\,(k \cdot N\,p_1 \cdot h \cdot T(\Phi[F_1]\,h)) \\
= & \quad \{\text{naturality of } h :: T \cdot M \to N\text{, and functoriality of } T\} \\
& f' \cdot B\,id\,(k \cdot h \cdot T(M\,p_1 \cdot \Phi[F_1]\,h)) \\
= & \quad \{\textbf{assume } M\,p_1 \cdot \Phi[F_1]h = \Phi'[F_1](k \cdot h)\} \\
& f' \cdot B\,id\,(k \cdot h \cdot T(\Phi'[F_1](k \cdot h))) \\
= & \quad \{\text{definition}\} \\
& \Psi'\,(k \cdot h).
\end{array}
$$

The second assumption in this calculation, namely,

$$M\,p_1 \cdot \Phi[F_1]h \quad = \quad \Phi'[F_1](k \cdot h),$$

where $p_1 :: F_1\,N \cdot M' \to M' \cdot F_1\,N'$, is elaborated to simpler assumptions, using the inductive definitions of Φ and Φ'. We will spell out the details for our three examples.

Example 5.4 For *List* we have $\Phi[F_1]h = g :: M \to M$. Hence

FOLD-FUSION LAW FOR *List*. Given the typings $f :: \mathit{Base} \cdot \langle M, N \cdot \mathit{Pair}\rangle \to N$, $g :: M \to M$ and $p :: M' \to M'$, we have

$$k \cdot \mathit{gfold}\,f\,g = \mathit{gfold}\,f'\,(M\,p \cdot g) \quad \Leftarrow \quad k \cdot f = f' \cdot \mathit{base}\,id\,(k \cdot N\,p).$$

For example, consider the composition

$$\mathit{gfold}\,f\,id \cdot \mathit{gfold}\,\alpha\,g,$$

where $f :: \mathit{Base} \cdot \langle M, N \cdot \mathit{Pair}\rangle \to N$ and $g :: M \to M$. The function $\mathit{gfold}\,f\,id :: \mathit{List} \cdot M \to N$ is a fold, while the function $\mathit{gfold}\,\alpha\,g :: \mathit{List} \cdot M \to \mathit{List} \cdot M$ is called the *triangle* with respect to g. We have

$$\mathit{gfold}\,f\,id \cdot \alpha \quad = \quad f \cdot \mathit{base}\,id\,(\mathit{gfold}\,f\,id \cdot \mathit{list}\,id).$$

Hence an appeal to fold-fusion gives

$$\mathit{gfold}\,f\,id \cdot \mathit{gfold}\,\alpha\,g \quad = \quad \mathit{gfold}\,f\,g.$$

Thus every generalised fold on lists (and indeed any regular type) can be factored as a regular fold after a triangle. In fact, every triangle can be expressed as a regular fold:

$$
\begin{array}{ll}
& \mathit{gfold}\,\alpha\,g \cdot \alpha \\
= & \quad \{\text{definition}\} \\
& \alpha \cdot \mathit{base}\,id\,(\mathit{gfold}\,\alpha\,g \cdot \mathit{list}\,g) \\
= & \quad \{\text{map-fusion}\} \\
& \alpha \cdot \mathit{base}\,id\,(\mathit{gfold}\,(\alpha \cdot \mathit{base}\,g\,id)\,g) \\
= & \quad \{\text{fold-fusion (backwards)}\} \\
& \alpha \cdot \mathit{base}\,id\,(g \cdot \mathit{gfold}\,\alpha\,g) \\
= & \quad \{\text{functoriality of } \mathit{base}\} \\
& \alpha \cdot \mathit{base}\,id\,g \cdot \mathit{base}\,id\,(\mathit{gfold}\,\alpha\,g).
\end{array}
$$

Hence $gfold\,\alpha\,g = fold\,(\alpha \cdot base\,g\,id)$, and $gfold\,f\,g$ is a composition of regular folds. Note that we have not used any property of *base* except functoriality, so this factorisation may be applied to the generalised fold on any regular type.

Example 5.5 For *Nest*, we have $\Phi[F_1]h = g :: Pair \cdot M \to M \cdot Pair$. Hence

FOLD-FUSION LAW FOR *Nest*. Given the typings

$$\begin{array}{lll} f & :: & Base \cdot \langle M, N \cdot Pair \rangle \to N \\ g & :: & Pair \cdot M \to M \cdot Pair \\ p & :: & Pair \cdot M' \to M' \cdot Pair \end{array}$$

we have

$$k \cdot gfold\,f\,g = gfold\,f'\,(M\,p \cdot g) \quad \Leftarrow \quad k \cdot f = f' \cdot base\,id\,(k \cdot N\,p).$$

The fold-fusion law for *Nest* takes the same form as for *List*, Though the types of g and p are different. Here is a simple example. The function $listify :: Nest \to List$ can be defined by

$$listify \quad = \quad gfold\,(\alpha \cdot base\,id\,(concat \cdot list\,duo))\,id$$

where $\alpha :: Base \cdot \langle Id, List \rangle \to List$ is the initial list algebra, $duo :: Pair \to List$ converts a pair of values into a list of two values, and $concat :: List \cdot List \to List$ concatenates a list of lists into one list. Here $M = Id$ and $N = List$.

Furthermore, the function $sumL :: List \cdot \underline{Int} \to \underline{Int}$, which sums a list of integers, is given by

$$sumL \quad = \quad fold\,sumB,$$

where *fold* is the fold function for lists.

Now consider the combination $sumL \cdot listify$. We can fuse the two functions provided

$$sumL \cdot \alpha \cdot base\,id\,(concat \cdot list\,duo) \quad = \quad f' \cdot base\,id\,(sumL \cdot list\,p),$$

for some $p :: Pair \cdot \underline{Int} \to \underline{Int}$ and $f' :: Base \cdot \langle \underline{Int}, \underline{Int} \rangle \to \underline{Int}$. For the proviso we can calculate:

$$\begin{array}{ll} & sumL \cdot \alpha \cdot base\,id\,(concat \cdot list\,duo) \\ = & \quad \{\text{definition of } sumL\} \\ & sumB \cdot base\,id\,sumL \cdot base\,id\,(concat \cdot list\,duo) \\ = & \quad \{\text{functoriality of } base\} \\ & sumB \cdot base\,id\,(sumL \cdot concat \cdot list\,duo) \\ = & \quad \{\text{property of } sumL \text{ and functoriality of } list\} \\ & sumB \cdot base\,id\,(sumL \cdot list\,(sumL \cdot duo)) \\ = & \quad \{\text{since } sumL \cdot duo = plus\} \\ & sumB \cdot base\,id\,(sumL \cdot list\,plus). \end{array}$$

Hence $sumL \cdot listify = gfold\,sumB\,plus$. The expression on the right is just the function $sumN$, given in the previous section, for summing a nest of integers directly.

Example 5.6 For $T = \mathit{Host}$, we have $\Phi[F_1]h = g_1 \cdot \mathit{id} \times (h \cdot T\, g_2)$, where $g_1 :: M \times N \to M \cdot (\mathit{Id} \times N)$ and $g_2 :: M \to M$. For fold-fusion we require $k \cdot f = f' \cdot \mathit{base}\, \mathit{id}\, (k \cdot N\, p)$, where $p :: M' \times (N \cdot M') \to M' \cdot (\mathit{Id} \times N')$ is required to satisfy the equation

$$M\, p \cdot \Phi[F_1]h \quad = \quad \Phi'[F_1](k \cdot h).$$

This condition expands to

$$M\, p \cdot g_1 \cdot \mathit{id} \times (h \cdot T\, g_2) \quad = \quad g_1' \cdot \mathit{id} \times (k \cdot h \cdot T\, g_2').$$

To eliminate the dependence on h, suppose

$$M\, p \cdot g_1 \quad = \quad g_1' \cdot \mathit{id} \times (k \cdot N\, q)$$

for some $q :: M' \to M'$. Then we can argue:

$$\begin{array}{cl} & M\, p \cdot g_1 \cdot \mathit{id} \times (h \cdot T\, g_2) \\ = & \quad \{\text{assumption, and functoriality of } \times\} \\ & g_1' \cdot \mathit{id} \times (k \cdot N\, q \cdot h \cdot T\, g_2) \\ = & \quad \{\text{naturality of } h :: T \cdot M \to N \text{ and functoriality of } T\} \\ & g_1' \cdot \mathit{id} \times (k \cdot h \cdot T(M\, q \cdot g_2)) \end{array}$$

Hence the fold-fusion law can be put in the form

Fold-fusion law for *Host*. Given the typings

$$\begin{array}{rcl} f & :: & \mathit{Base} \cdot \langle M, N \cdot (\mathit{Id} \times N)\rangle \to N \\ g_1 & :: & M \times N \to M \cdot (\mathit{Id} \times N) \\ g_2 & :: & M \to M \\ q & :: & M' \to M', \end{array}$$

we have

$$k \cdot \mathit{gfold}\, f\, g_1\, g_2 \quad = \quad \mathit{gfold}\, f'\, g_1'\, (M\, q \cdot g_2)$$

provided that, for some $p :: M' \times (N \cdot M') \to M' \cdot (\mathit{Id} \times N')$,

$$k \cdot f = f' \cdot \mathit{base}\, \mathit{id}\, (k \cdot N\, p) \quad \text{and} \quad M\, p \cdot g_1 = g_1' \cdot \mathit{id} \times (k \cdot N\, q).$$

6. Uniqueness of generalised folds

Our aim in this section is to show that generalised folds are the unique solutions of their defining equations, which have the form

$$\mathit{gfold}\, f\, g_1 \cdots g_m \cdot \alpha_M \quad = \quad \Psi\, (\mathit{gfold}\, f\, g_1 \cdots g_m)$$

for some form $\Psi :: \forall X. (X \cdot M \to N) \to (F\, X \cdot M \to N)$. It turns out that it is easier to solve a more general problem. Suppose we are given functors $F :: \mathbf{C} \to \mathbf{C}$ and $L :: \mathbf{C} \to \mathbf{D}$ for categories $\mathbf{C}$ and $\mathbf{D}$, for which F has least fixed point $\alpha :: F\, T \to T$, and a natural transformation $\Psi :: \forall A. (L\, A \to B) \to (L\, (F\, A) \to B)$. We seek conditions on F and L to ensure there is a unique $x :: L\, T \to B$ such that

$$x \cdot L\, \alpha \quad = \quad \Psi\, x. \tag{1}$$

Note that the naturality of Ψ implies that

$$\Psi\,(f \cdot L\,x) \quad = \quad \Psi\,f \cdot L\,(F\,x). \tag{2}$$

Some instances of this general scheme are:

- The *gfold* over nested types is the special case where F and L are higher order functors, with $L = (\cdot M)$. The action of this functor on natural transformations, like α, yields instances α_M.
- In the special case $L = Id$, equation (2) implies

  $$\Psi\,x \quad = \quad \Psi\,id \cdot F\,x$$

  so that Ψ is determined by $\Psi\,id :: F\,B \to B$ and the generalised fold reduces to an ordinary fold.
- Further instances are defining equations of the form

  $$x \cdot L\,\alpha \quad = \quad f \cdot G\,x \cdot g$$

  for functions $f :: G\,B \to B$ and $g :: L \cdot F \to G \cdot L$. For example, *zip*-like functions [FSZ94] are of this form, with $L = \times$ (a binary functor) and $g :: F_1 \times F_2 \to G \cdot \times$.

Furthermore, this uniqueness property leads to a general fusion law, stating that for any natural transformation $\Phi :: \forall\,A.\,(L\,A \to B) \to (L'\,A \to B)$,

$$\Phi\,(gfold\,\Psi) = gfold\,\Psi' \quad \Leftarrow \quad \Phi \cdot \Psi = \Psi' \cdot \Phi.$$

All the previous fusion laws are instances of this scheme.

We present two solutions to this problem. The first depends on the details of the colimit construction of the fixed point, but works for a larger class of functors L than the second, more abstract, approach. For example, the first approach works for $L = \times$, while the second does not.

6.1. First method: colimits

Theorem 1. Suppose the functors F and L preserve colimits of chains (all those we can define in Haskell do), and L preserves initiality. Then equation (1) has a unique solution.

Proof. Suppose the colimit used to construct the fixed point T of F consists of functions $e_n :: F^n\,0 \to T$. By construction, the isomorphism $\alpha :: F\,T \cong T$ satisfies

$$\alpha \cdot F\,e_n \quad = \quad e_{n+1} \tag{3}$$

for each n. Since L preserves initiality, there is a unique arrow $z :: L\,0 \to B$. We shall show that

$$x \cdot L\,\alpha = \Psi\,x \quad \equiv \quad \forall\,n.\,x \cdot L\,e_n = \Psi^n\,z.$$

Since the arrows $L\,e_n$ comprise a colimit, this establishes the existence and uniqueness of x.

First, we prove

$$\forall\,n.\,x \cdot L\,e_n = \Psi^n\,z \quad \Leftarrow \quad x \cdot L\,\alpha = \Psi\,x$$

by induction on n. The base case is immediate from the initiality of $L\,0$. The induction step is

$$\begin{array}{ll}
 & x \cdot L\,e_{n+1} \\
= & \quad \{\text{equation (3)}\} \\
 & x \cdot L\,(\alpha \cdot F\,e_n) \\
= & \quad \{\text{functor}\} \\
 & x \cdot L\,\alpha \cdot L\,(F\,e_n) \\
= & \quad \{\text{hypothesis}\} \\
 & \Psi\,x \cdot L\,(F\,e_n) \\
= & \quad \{\text{equation (2)}\} \\
 & \Psi\,(x \cdot L\,e_n) \\
= & \quad \{\text{induction hypothesis}\} \\
 & \Psi\,(\Psi^n\,z)
\end{array}$$

To establish the reverse implication, we reason

$$\begin{array}{ll}
 & x \cdot L\,\alpha = \Psi\,x \\
\equiv & \quad \{\text{isomorphism}\} \\
 & x = \Psi\,x \cdot L\,\alpha^{-1} \\
\Leftarrow & \quad \{\text{colimit}\} \\
 & \forall\,n.\,\Psi\,x \cdot L\,\alpha^{-1} \cdot L\,e_n = \Psi^n\,z \\
\Leftarrow & \quad \{\text{by cases of } n \text{ (see below)}\} \\
 & \forall\,n.\,x \cdot L\,e_n = \Psi^n\,z
\end{array}$$

The case $n = 0$ is immediate from the initiality of $L\,0$. The case $n = m + 1$ is

$$\begin{array}{ll}
 & \Psi\,x \cdot L\,\alpha^{-1} \cdot L\,e_{m+1} \\
= & \quad \{\text{equation (3)}\} \\
 & \Psi\,x \cdot L\,\alpha^{-1} \cdot L\,\alpha \cdot L\,(F\,e_m) \\
= & \quad \{\text{functor, isomorphism}\} \\
 & \Psi\,x \cdot L\,(F\,e_m) \\
= & \quad \{\text{equation (2)}\} \\
 & \Psi\,(x \cdot L\,e_m) \\
= & \quad \{\text{hypothesis}\} \\
 & \Psi\,(\Psi^m\,z)
\end{array}$$

completing the proof. □

6.2. Second method: adjoints

Another approach is to note that we seek a function of type $L\,T \to B$, whereas a fold can supply a function of type $T \to B'$. Recall that an adjunction between L and a functor R defines an isomorphism

$$\varphi \quad :: \quad \forall\,A, B.\,(L\,A \to B) \cong (A \to R\,B). \tag{4}$$

Setting $A = T$, this yields a correspondence between generalised folds and ordinary folds.

Theorem 2. If L has a right adjoint, then equation (1) has a unique solution.

Proof. Let φ denote the isomorphism defined by the adjunction, as above. We will be applying it to both sides of equation (1). We calculate

$$\begin{array}{ll}
& \varphi\,(\Psi\,x) \\
= & \quad \{\text{isomorphism}\} \\
& \varphi\,(\Psi\,(\varphi^{-1}\,(\varphi\,x))) \\
= & \quad \{\text{naturality of } \varphi^{-1}\text{, introducing the counit } \varepsilon = \varphi^{-1}\,id\} \\
& \varphi\,(\Psi\,(\varepsilon \cdot L\,(\varphi\,x))) \\
= & \quad \{\text{naturality of } \Psi\} \\
& \varphi\,(\Psi\,\varepsilon \cdot L\,(F\,(\varphi\,x))) \\
= & \quad \{\text{naturality of } \varphi\} \\
& \varphi\,(\Psi\,\varepsilon) \cdot F\,(\varphi\,x)
\end{array}$$

Thus we have

$$\begin{array}{ll}
& x \cdot L\,\alpha = \Psi\,x \\
\equiv & \quad \{\text{isomorphism}\} \\
& \varphi\,(x \cdot L\,\alpha) = \varphi\,(\Psi\,x) \\
\equiv & \quad \{\text{naturality of } \varphi\} \\
& \varphi\,x \cdot \alpha = \varphi\,(\Psi\,x) \\
\equiv & \quad \{\text{above calculation}\} \\
& \varphi\,x \cdot \alpha = \varphi\,(\Psi\,\varepsilon) \cdot F\,(\varphi\,x) \\
\equiv & \quad \{\text{fold uniqueness}\} \\
& \varphi\,x = \mathit{fold}\,(\varphi\,(\Psi\,\varepsilon)) \\
\equiv & \quad \{\text{isomorphism}\} \\
& x = \varphi^{-1}\,(\mathit{fold}\,(\varphi\,(\Psi\,\varepsilon)))
\end{array}$$

□

Note that this version is strictly weaker than the lower-level approach using colimits: any functor with a right adjoint necessarily preserves all colimits, while $\times$ is an example of a functor that preserves initiality and colimits of chains, but not coproducts.

To apply this theorem to generalised folds over nested datatypes, it remains to construct a right adjoint to $(\cdot M)$. This is a well-studied problem in category theory, where it is known as the right Kan extension [Mac71, X.3], and is known to exist if M is defined on a small complete category. It is also possible to eliminate the size condition by restricting the class of functors M, say to nested functors.

We can define this adjunction in Haskell by introducing a *continuation* type $\mathit{Cont}\,M\,N$, and constructing an isomorphism pair between $T \cdot M \rightarrow N$ and $T \rightarrow \mathit{Cont}\,M\,N$. The functor Cont is declared as

$$\textbf{newtype}\ \mathit{Cont}\,m\,n\,a \quad = \quad \mathit{MkCont}\,(\forall\,b.\,(a \rightarrow m\,b) \rightarrow n\,b)$$

Thus, $Cont$ wraps a higher-order polymorphic function. For fixed M and N, the functorial action of $Cont$ is defined by

$$\begin{array}{lcl} cont & :: & (a \to b) \to Cont\; m\; n\; a \to Cont\; m\; n\; b \\ cont\; f\; (MkCont\; g) & = & MkCont\; (\lambda\, k.\, g(k \cdot f)) \end{array}$$

One half of the isomorphism is the function

$$\begin{array}{lcl} toCont & :: & Functor\; t \Rightarrow (\forall\, a.\, t\; (m\; a) \to n\; a) \to t\; b \to Cont\; m\; n\; b \\ toCont\; f\; x & = & MkCont\; (\lambda\, k.\, f\; (map_t\; k\; x)) \end{array}$$

The function map_t is the functorial action of t. The subscript is added for clarity; it is not used in Haskell.

The other half of the isomorphism pair is

$$\begin{array}{lcl} fromCont & :: & Functor\; t \Rightarrow (\forall\, a.\, t\; a \to Cont\; m\; n\; a) \to t\; (m\; b) \to n\; b \\ fromCont\; f & = & applyto\; id \cdot f \end{array}$$

where the function $applyto\; k$ applies a continuation to k:

$$\begin{array}{lcl} applyto & :: & (a \to m\; b) \to Cont\; m\; n\; a \to n\; b \\ applyto\; k\; f & = & unCont\; f\; k \\ & & \\ unCont & :: & Cont\; m\; n\; a \to (a \to m\; b) \to n\; b \\ unCont\; (MkCont\; x) & = & x \end{array}$$

Then $toCont$ and $fromCont$ is an isomorphism pair, meaning that

$$\begin{array}{lcl} toCont \cdot fromCont & = & id \\ fromCont \cdot toCont & = & id. \end{array}$$

The first identity is verified by the following calculation:

$$\begin{array}{cl} & toCont\; (fromCont\; f)\; x \\ = & \quad \{\text{definitions of } toCont \text{ and } fromCont\} \\ & MkCont\; (\lambda\, k.\, applyto\; id\; (f\; (map_t\; k\; x))) \\ = & \quad \{\text{definition of } applyto\} \\ & MkCont\; (\lambda\, k.\, unCont\; (f\; (map_t\; k\; x))\; id) \\ = & \quad \{\text{naturality of } f\} \\ & MkCont\; (\lambda\, k.\, unCont\; (cont\; k\; (f\; x))\; id) \\ = & \quad \{\text{definition of } cont\} \\ & MkCont\; (\lambda\, k.\, (\lambda\, k'.\, unCont\; (f\; x)\; (k' \cdot k))\; id) \\ = & \quad \{\text{beta reduction; identity}\} \\ & MkCont\; (\lambda\, k.\, unCont\; (f\; x)\; k) \\ = & \quad \{\text{eta reduction; definition of } unCont\} \\ & f\; x. \end{array}$$

For the second identity, we calculate:

$$\begin{array}{cl} & fromCont\; (toCont\; f)\; x \\ = & \quad \{\text{definitions of } fromCont,\; applyto\} \\ & unCont\; (toCont\; f\; x)\; id \\ = & \quad \{\text{definitions of } toCont \text{ and } unCont\} \end{array}$$

$$
\begin{array}{ll}
& (\lambda\, k.\, f\ (map_t\ k\ x))\ id \\
= & \quad \{\text{beta reduction}\} \\
& f\ (map_t\ id\ x) \\
= & \quad \{\text{functoriality of } map_t\} \\
& f\ x.
\end{array}
$$

7. Final remarks

The subject of nested datatypes and what they might provide for the practical programmer is still in its infancy. Most of the published work so far (in particular, [Oka98]) uses nested types only as a conceptual tool; the results are translated into programming terms by embedding nested types in regular ones in a systematic way. One reason that nested types have not been used directly is that, until recently, there has been no language support for defining functions over such types. Recent versions of Haskell permit rank-2 type signatures and, as we have seen, such a move is necessary to implement functions over nested types.

Defining functions over nested datatypes by explicit recursion is complicated, and even more error prone than using a similar style with regular types. The real purpose of concentrating on the fold function for a datatype is that it provides a structured approach to inductive functional programming. Such a concentration has added force when the datatype is nested.

Acknowledgements

Peter Freyd, Andy Pitts, James Worrell and Dominic Hughes directed us to the appropriate category theory for the adjunction in Section 6.2. Anonymous referees made several helpful suggestions.

References

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

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, second edition, 1998.

[BM98] Richard S. Bird and Lambert Meertens. Nested datatypes. In *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[BP99] Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.

[CL95] R.H. Connelly and F. Lockwood Morris. A generalisation of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, 1995.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, June 1994.

[Hag87] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, UK, 1987. Technical Report ECS-LFCS-87-38.

[Hoo97] Paul Hoogendijk. *A Generic Theory of Data Types.* PhD thesis, Eindhoven University of Technology, 1997.

[Jon98] Mark Jones. A technical summary of the new features in Hugs 1.3c, 1998. unpublished.

[JS93] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 208–232. Springer, 1993.

[Lam70] J. Lambek. Subequalizers. *Canadian Mathematical Bulletin*, 13:337–349, 1970.

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

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York, 1971.

[Mal90a] Grant R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, 1990.

[Mal90b] Grant R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3):255–279, 1990.

[McC84] N. J. McCracken. The typechecking of programs with implicit type structure. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 1984.

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

[PL97] S. Peyton Jones and J. Launchbury. Explicit quantification in Haskell, 1997. See: `http://www.dcs.gla.ac.uk/people/personal/simonpj/`.