Nested Datatypes

Richard Bird[1] and Lambert Meertens[2]

[1] Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
bird@comlab.ox.ac.uk
[2] CWI and Department of Computer Science, Utrecht University,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
lambert@cwi.nl

Abstract. A nested datatype, also known as a *non-regular* datatype, is a parametrised datatype whose declaration involves different instances of the accompanying type parameters. Nested datatypes have been mostly ignored in functional programming until recently, but they are turning out to be both theoretically important and useful in practice. The aim of this paper is to suggest a functorial semantics for such datatypes, with an associated calculational theory that mirrors and extends the standard theory for regular datatypes. Though elegant and generic, the proposed approach appears more limited than one would like, and some of the limitations are discussed.

> Hark, by the bird's song ye may learn the nest.
> TENNYSON *The Marriage of Geraint*

1 Introduction

Consider the following three datatype definitions, all of which are legal Haskell declarations:

$$\begin{aligned}
&\textbf{data}\ \mathit{List}\ a\ = \mathit{NilL} \mid \mathit{ConsL}\,(a, \mathit{List}\ a)\\
&\textbf{data}\ \mathit{Nest}\ a = \mathit{NilN} \mid \mathit{ConsN}\,(a, \mathit{Nest}\,(a, a))\\
&\textbf{data}\ \mathit{Bush}\ a = \mathit{NilB} \mid \mathit{ConsB}\,(a, \mathit{Bush}\,(\mathit{Bush}\ a))
\end{aligned}$$

The first type, $\mathit{List}\ a$, describes the familiar type of cons-lists. Elements of the second type $\mathit{Nest}\ a$ are like cons-lists, but the lists are not homogeneous: each step down the list, entries are "squared". For example, using brackets and commas instead of the constructors NilN and ConsN, one value of type $\mathit{Nest}\,\mathit{Int}$ is

$$[7,\ (1,2),\ ((6,7),(7,4)),\ (((2,5),(7,1)),((3,8),(9,3)))]$$

This nest has four entries which, taken together, contain fifteen integers.

In the third type $\mathit{Bush}\ a$, at each step down the list, entries are "bushed". For example, one value of type $\mathit{Bush}\,\mathit{Int}$ is

$$
\begin{array}{l}
[\,4, \\
\quad [\,8, [\,5\,], [\,[\,3\,]\,]\,], \\
\quad [\,[\,7\,], [\,], [\,[\,[\,7\,]\,]\,]\,], \\
\quad [\,[\,[\,], [\,[\,0\,]\,]\,]\,] \\
]
\end{array}
$$

This bush contains four entries, the first of which is an element of *Int*, the second an element of *Bush Int*, the third an element of *Bush* (*Bush Int*), and so on. In general, the n-th entry (counting from 0) of a list of type *Bush a* has type $Bush^n\ a$.

The datatype *List a* is an example of a so-called *regular* datatype, while *Nest a* and *Bush a* are examples of *non-regular* datatypes. Mycroft [17] calls such schemes *polymorphic recursions*. We prefer the term *nested* datatypes. In a regular datatype declaration, occurrences of the declared type on the right-hand side of the defining equation are restricted to copies of the left-hand side, so the recursion is "tail recursive". In a nested datatype declaration, occurrences of the datatype on the right-hand side appear with different instances of the accompanying type parameter(s), so the recursion is "nested".

In a language like Haskell or ML, with a Hindley-Milner type discipline, it is simply not possible to define all the useful functions one would like over a nested datatype, even though such datatype declarations are themselves perfectly legal. This remark applies even to recent extensions of such languages (in particular, Haskell 1.4), in which one is allowed to declare the types of problematic functions, and to use the type system for checking rather than inferring types. To be sure, a larger class of functions can now be defined, but one still cannot define important generic functions, such as *fold*, over nested types.

On the other hand, the most recent versions of Hugs and GHC (the Glasgow Haskell Compiler) both support so-called *rank-2* type signatures, in which one can universally quantify over type constructors as well as types (see [20]). By using such signatures one can construct most of the functions over nested datatypes that one wants. We will return to this point below. However, rank-2 type signatures are not yet part of standard Haskell.

The upshot of the current situation is that nested datatypes have been rather neglected in functional programming. However, they are conceptually important and evidence is emerging (e.g. [3, 18, 19]) of their usefulness in functional data structure design. A brief illustration of what they can offer is given in Section 2.

Regular datatypes, on the other hand, are the bread and butter of functional programming. Recent work on *polytypic* programming (e.g. [2, 9, 15]) has systematised the mathematics of program construction with regular datatypes by focusing on a small number of generic operators, such as *fold*, that can be defined for all such types. The basic idea, reviewed below, is to define a regular datatype as an initial object in a category of F-algebras for an appropriate functor F. Indeed, this idea appeared much earlier in the categorical literature, for instance in [10]. As a consequence, polytypic programs are parametrised by one or more regular functors. Different instances of these functors yield the concrete programs we know and love.

The main aim of this paper is to investigate what form an appropriate functorial semantics for nested datatypes might take, thereby putting more 'poly' into 'polytypic'. The most appealing idea is to replace first-order functors with higher-order functors over functor categories. In part, the calculational theory remains much the same. However, there are limitations with this approach, in that some expressive power seems to be lost, and some care is needed in order that the standard functorial semantics of regular datatypes may be recovered as a special case. It is important to note that we will not consider datatype declarations containing function spaces in this paper; see [6, 16] for ways of dealing with function spaces in datatype declarations.

2 An example

Let us begin with a small example to show the potential of nested datatypes. The example was suggested to us by Oege de Moor. In the De Bruijn notation for lambda expressions, bound variables introduced by lambda abstractions are represented by natural numbers. An occurrence of a number n in an expression represents the bound variable introduced by the n-th nested lambda abstraction. For example, $\underline{0}\,(\underline{1}\,\underline{1})$ represents the lambda term

$$\lambda\, x.\lambda y.x\,(y\;y)$$

On the other hand, $\underline{0}\,(w\,\underline{1})$ represents the lambda term

$$\lambda\, x.\lambda y.x\,(w\;y)$$

in which w is a free variable.

One way to capture this scheme is to use a nested datatype:

$$\begin{array}{l} \textbf{data}\ \mathit{Term}\ a = \mathit{Var}\ a \mid \mathit{App}\,(\mathit{Term}\ a, \mathit{Term}\ a) \mid \mathit{Abs}\,(\mathit{Term}\,(\mathit{Bind}\ a)) \\ \textbf{data}\ \mathit{Bind}\ a\ \ = \mathit{Zero} \mid \mathit{Succ}\ a \end{array}$$

Elements of $\mathit{Term}\,a$ are either free variables (of type $\mathit{Var}\,a$), applications, or abstractions. In an abstraction, the outermost bound variable is represented by $\mathit{Var\ Zero}$, the next by $\mathit{Var}\,(\mathit{Succ\ Zero})$, and so on. Free variables in an abstraction containing n nested bindings have type $\mathit{Var}\,(\mathit{Succ}^n\,a)$. The type $\mathit{Term}\,a$ is nested because $\mathit{Bind}\,a$ appears as a parameter of Term on the right-hand side of the declaration.

For example, $\lambda\,x.\lambda y.x\,(w\;y)$ may be represented by the following term of type $\mathit{Term\ Char}$:

$$\mathit{Abs}\,(\mathit{Abs}\,(\mathit{App}\,(\mathit{Var\ Zero},\ \mathit{App}\,(\mathit{Var}\,(\mathit{Succ}\,(\mathit{Succ}\,\text{'w'})),\ \mathit{Var}\,(\mathit{Succ\ Zero})))))$$

The closed lambda terms – those containing no free variables – are elements of $\mathit{Term\ Empty}$, where Empty is the empty type containing no members.

The function *abstract*, which takes a term and a variable and abstracts over that variable, can be defined in the following way:

$$\begin{array}{lcl} \mathit{abstract} & :: & (\mathit{Term}\,a, a) \rightarrow \mathit{Term}\,a \\ \mathit{abstract}\,(t, x) & = & \mathit{Abs}\,(\mathit{lift}\,(t, x)) \end{array}$$

The function *lift* is defined by

$$
\begin{array}{ll}
\mathit{lift} & :: (\mathit{Term}\,a, a) \to \mathit{Term}\,(\mathit{Bind}\,a)\\
\mathit{lift}\,(\mathit{Var}\,y, x) & = \textbf{if } x = y \textbf{ then } \mathit{Var}\,\mathit{Zero} \textbf{ else } \mathit{Var}\,(\mathit{Succ}\,y)\\
\mathit{lift}\,(\mathit{App}\,(u,v), x) & = \mathit{App}\,(\mathit{lift}\,(u,x), \mathit{lift}\,(v,x))\\
\mathit{lift}\,(\mathit{Abs}\,t, x) & = \mathit{Abs}\,(\mathit{lift}\,(t, \mathit{Succ}\,x))
\end{array}
$$

The β-reduction of a term is implemented by

$$
\begin{array}{ll}
\mathit{reduce} & :: (\mathit{Term}\,a, \mathit{Term}\,a) \to \mathit{Term}\,a\\
\mathit{reduce}\,(\mathit{Abs}\,s, t) & = \mathit{subst}\,(s,t)
\end{array}
$$

where

$$
\begin{array}{ll}
\mathit{subst} & :: (\mathit{Term}\,(\mathit{Bind}\,a), \mathit{Term}\,a) \to \mathit{Term}\,a\\
\mathit{subst}\,(\mathit{Var}\,\mathit{Zero}, t) & = t\\
\mathit{subst}\,(\mathit{Var}\,(\mathit{Succ}\,x), t) & = \mathit{Var}\,x\\
\mathit{subst}\,(\mathit{App}\,(u,v), t) & = \mathit{App}\,(\mathit{subst}\,(u,t), \mathit{subst}\,(v,t))\\
\mathit{subst}\,(\mathit{Abs}\,s, t) & = \mathit{Abs}\,(\mathit{subst}\,(s, \mathit{term}\,\mathit{Succ}\,t))
\end{array}
$$

The function $\mathit{term}\,f$ maps f over a term:

$$
\begin{array}{ll}
\mathit{term} & :: (a \to b) \to (\mathit{Term}\,a \to \mathit{Term}\,b)\\
\mathit{term}\,f\,(\mathit{Var}\,x) & = \mathit{Var}\,(f\,x)\\
\mathit{term}\,f\,(\mathit{App}\,(u,v)) & = \mathit{App}\,(\mathit{term}\,f\,u, \mathit{term}\,f\,v)\\
\mathit{term}\,f\,(\mathit{Abs}\,t) & = \mathit{Abs}\,(\mathit{term}\,(\mathit{bind}\,f)\,t)
\end{array}
$$

The subsidiary function $\mathit{bind}\,f$ maps f over elements of $\mathit{Bind}\,a$:

$$
\begin{array}{ll}
\mathit{bind} & :: (a \to b) \to (\mathit{Bind}\,a \to \mathit{Bind}\,b)\\
\mathit{bind}\,f\,\mathit{Zero} & = \mathit{Zero}\\
\mathit{bind}\,f\,(\mathit{Succ}\,x) & = \mathit{Succ}\,(f\,x)
\end{array}
$$

It is a routine induction to show that

$$\mathit{reduce}\,(\mathit{abstract}\,(t,x), \mathit{Var}\,x) = t$$

for all terms t of type $\mathit{Term}\,a$ and all x of type a.

Modulo the requirement that a and $\mathit{Bind}\,a$ be declared as equality types (because elements are compared for equality in the definition of *lift*) the programs above are acceptable to Haskell 1.4, provided the type signatures are included as part of the definitions.

3 Datatypes as initial algebras

The standard semantics (see e.g. [8, 10]) of inductive datatypes parametrised by n type parameters employs functors of type $\mathbf{C} \times \cdots \times \mathbf{C} \to \mathbf{C}$, where the product has $n+1$ occurrences of $\mathbf{C}$. For simplicity, we will consider only the case $n = 1$. The category $\mathbf{C}$ cannot be arbitrary: essentially, it has to contain finite sums and products, and colimits of all ascending chains. The category **Fun** (also known as **Set**), whose objects are sets and whose arrows are typed total functions, has everything needed to make the theory work.

To illustrate, the declaration of *List* as a datatype is associated with a binary functor F whose action on objects of $\mathbf{C} \times \mathbf{C}$ is defined by

$$F(a, b) = 1 + a \times b$$

Introducing the unary functor F_a, where $F_a(b) = F(a, b)$, the declaration of $List\ a$ can now be rewritten in the form

$$\mathbf{data}\ List\ a \xleftarrow{\alpha_a} F_a(List\ a)$$

in which $\alpha_a :: F_a(List\ a) \to List\ a$. For the particular functor F associated with *List*, the arrow α_a takes the form $(NilL_a, ConsL_a)$, where $NilL_a :: 1 \to List\ a$ and $ConsL_a :: a \times List\ a \to List\ a$. This declaration can can be interpreted as the assertion that the arrow α_a and the object $List\ a$ are the "least" values with this typing. More precisely, given any arrow

$$f :: F_a(b) \to b$$

the assertion is that there is a unique arrow $h :: List\ a \to b$ satisfying the equation

$$h \cdot \alpha_a = f \cdot F(id_a, h)$$

The unique arrow h is denoted by $fold\ f$. The arrow h is also called a *catamorphism*, and the notation $(\![f]\!)$ is also used for $fold\ f$. In algebraic terms, $List\ a$ is the carrier of the *initial* algebra α_a of the functor F_a and $fold\ f$ is the unique F_a-homomorphism from the initial algebra to f.

A surprising number of consequences flow from this characterisation. In particular, $fold\ \alpha_a$ is the identity arrow on $List\ a$. Also, one can show that α_a is an isomorphism, with inverse $fold\ (F(id_a, \alpha_a))$. As a result, one can interpret the declaration of *List* as the assertion that, up to isomorphism, $List\ a$ is the least fixed point of the equation $x = F(a, x)$.

The type constructor *List* can itself can be made into a functor by defining its action on an arrow $f : a \to b$ by

$$list\ f = fold\ (\alpha_b \cdot F(f, id))$$

In functional programming $list\ f$ is written $map\ f$. Expanding the definition of *fold*, we have

$$list\ f \cdot \alpha_a = \alpha_b \cdot F(f, list\ f)$$

This equation states that α is a natural transformation of type $\alpha :: G \to List$, where $G\ a = F(a, List\ a)$.

The most important consequence of the characterisation is that it allows one to introduce new functions by structural recursion over a datatype. As a simple example, $fold\ (zero, plus)$ sums the elements of a list of numbers.

Functors built from constant functors, type functors (like *List*), the identity and projection functors, using coproduct, product, and composition operations, are called *regular* functors. For further details of the approach, consult, e.g., [12] or [1].

For *Nest* and *Bush* the theory above breaks down. For example, introducing $Q\ a = a \times a$ for the squaring functor, the corresponding functorial declaration for *Nest* would be

$$\mathbf{data}\ \mathit{Nest}\ a \xleftarrow{\alpha_a} F(a, \mathit{Nest}\,(Q\ a))$$

where F is as before, and α_a applies *NilN* to left components and *ConsN* to right components. However, it is not clear over what class of algebras α_a can be asserted to be initial.

4 A higher-order semantics

There is an appealing semantics for dealing with datatypes such as *Nest* and *Bush*, which, however, has certain limitations. We will give the scheme, then point out the limitations, and then give an alternative scheme that overcomes some of them.

The idea is to use higher-order functors of type

$$\mathit{Nat}(\mathbf{C}) \to \mathit{Nat}(\mathbf{C}),$$

where $\mathit{Nat}(\mathbf{C})$ is the category whose objects are functors of type $\mathbf{C} \to \mathbf{C}$ and whose arrows are natural transformations. We will use calligraphic letters for higher-order functors, and small Greek letters for natural transformations. Again, the category $\mathbf{C}$ cannot be arbitrary, but taking $\mathbf{C} = \mathbf{Fun}$ gives everything one needs. Here are three examples.

Example 1. The declaration of *List* can be associated with a higher-order functor $\mathcal{F}$ defined on objects (functors) by

$$\mathcal{F}(F)(a) = 1 + a \times F(a)$$
$$\mathcal{F}(F)(f) = \mathit{id}_1 + f \times F(f)$$

These equations define $\mathcal{F}(F)$ to be a functor for each functor F. The functor $\mathcal{F}$ can be expressed more briefly in the form

$$\mathcal{F}(F) = K1 + \mathit{Id} \times F$$

The constant functor $K\ a$ delivers the object a for all objects and the arrow id_a for all arrows, and Id denotes the identity functor. The coproduct $(+)$ and product $(\times)$ operations are applied pointwise.

The action of $\mathcal{F}$ on arrows (natural transformations) is defined in a similar style by

$$\mathcal{F}(\eta) = \mathit{id}_{K1} + \mathit{id} \times \eta$$

Here, id_{K1} delivers the identity arrow id_1 for each object of $\mathbf{C}$. If $\eta :: F \to G$, then $\mathcal{F}(\eta) :: \mathcal{F}(F) \to \mathcal{F}(G)$. We have $\mathcal{F}(\mathit{id}) = \mathit{id}$, and $\mathcal{F}(\eta \cdot \psi) = \mathcal{F}(\eta) \cdot \mathcal{F}(\psi)$, so $\mathcal{F}$ is itself a functor.

The previous declaration of *List* can now be written in the form

$$\textbf{data}\ \mathit{List} \xleftarrow{\alpha} \mathcal{F}(\mathit{List})$$

and interpreted as the assertion that α is the initial $\mathcal{F}$-algebra.

Example 2. The declaration of *Nest* is associated with a functor $\mathcal{F}$, defined on objects (functors) by

$$\begin{aligned}\mathcal{F}(F)(a) &= 1 + a \times F(Qa)\\ \mathcal{F}(F)(f) &= id_1 + f \times F(Qf)\end{aligned}$$

where Q is the squaring functor. More briefly,

$$\mathcal{F}(F) = K1 + Id \times (F \cdot Q)$$

where $F \cdot Q$ denotes the (functor) composition of F and Q. Where convenient, we will also write this composition as FQ for brevity.

The action of $\mathcal{F}$ on arrows (natural transformations) is defined by

$$\mathcal{F}(\eta) = id_{K1} + id \times \eta Q$$

where $\eta Q :: FQ \to GQ$ if $\eta :: F \to G$.

Example 3. The declaration of *Bush* is associated with a functor $\mathcal{F}$, defined on functors by

$$\mathcal{F}(F) = K1 + Id \times (F \cdot F)$$

and on natural transformations by

$$\mathcal{F}(\eta) = id_{K1} + id \times (\eta \star \eta)$$

The operator $\star$ denotes the horizontal composition of two natural transformations. If $\theta :: F \to G$ and $\psi :: H \to N$, then $\theta \star \psi :: FH \to GN$ is defined by $\theta \star \psi = \theta N \cdot F\psi$. In particular, if $\eta :: F \to G$, then $\eta \star \eta :: FF \to GG$.

Consider again the declaration of *Nest* given in the Introduction, and rewrite it in the form

$$\textbf{data}\ \mathit{Nest} \xleftarrow{\alpha} \mathcal{F}(\mathit{Nest})$$

The assertion that α is the initial $\mathcal{F}$-algebra means that for any arrow $\varphi :: \mathcal{F}(F) \to F$, there is a unique arrow $\theta :: \mathit{Nest} \to F$ satisfying the equation

$$\theta \cdot \alpha = \varphi \cdot \mathcal{F}(\theta).$$

The unique arrow θ is again denoted by $\mathit{fold}\,\varphi$.

We can express the equation above in Haskell. Note that for the particular functor $\mathcal{F}$ associated with *Nest*, the arrow φ takes the form $\varphi = (\varepsilon, \psi)$, where $\varepsilon :: K1 \to F$ and $\psi :: Id \times FQ \to F$. For any type a, the component ε_a is an arrow delivering a constant e of type $F\,a$, while ψ_a is an arrow f of type $(a, F(a,a)) \rightarrow F(a)$. Hence we can write

$$\begin{aligned} &\mathit{fold}\,(e,f)\,\mathit{NilN} &&= e \\ &\mathit{fold}\,(e,f)\,(\mathit{ConsN}\,(x,\mathit{xps})) &&= f(x, \mathit{fold}\,(e,f)\,\mathit{xps}) \end{aligned}$$

However, no principal type can be inferred for *fold* under the Hindley-Milner type discipline, so the use of *fold* in programs is denied us. Moreover, it is not possible to express the type of *fold* in any form that is acceptable to a standard Haskell type checker. On the other hand, in GHC (The Glasgow Haskell Compiler) one can declare the type of *fold* by using a rank-2 type signature:

$$\mathit{fold} :: (\forall f.\forall b.\,((\forall a.f\;a), (\forall a.(a, f(a,a)) \rightarrow f\;a)) \rightarrow \mathit{Nest}\;b \rightarrow f\;b)$$

This declaration uses both local universal quantification and abstraction over a type constructor. Such a signature is called a rank-2 type signature. With this asserted type, the function *fold* passes the GHC type-checker.

Observe that in the proposed functorial scheme, unlike the previous one for regular datatypes, the operator *fold* takes natural transformations to natural transformations. In particular, the fact that *Nest* is a functor is part of the assertion that *Nest* is the least fixed point of $\mathcal{F}$. The arrow $\mathit{nest}\,f$ cannot be defined as an instance of *fold* since it is not a natural transformation of the right type.

The typing $\alpha :: \mathcal{F}(\mathit{Nest}) \rightarrow \mathit{Nest}$ means that, given $f :: a \rightarrow b$, the following equation holds:

$$\mathit{nest}\,f \cdot \alpha_a = \alpha_b \cdot \mathcal{F}(\mathit{nest})\,f$$

We can express this equation at the point level by

$$\begin{aligned} &\mathit{nest}\,f\;\mathit{NilN} &&= \mathit{NilN} \\ &\mathit{nest}\,f\;(\mathit{ConsN}\,(x,\mathit{xps})) &&= \mathit{ConsN}\,(f\,x, \mathit{nest}\,(\mathit{square}\,f)\,\mathit{xps}) \end{aligned}$$

where $\mathit{square}\,f\,(x,y) = (f\,x, f\,y)$ is the action on arrows of the functor Q. The fact that *nest* is uniquely defined by these equations is therefore a consequence of the assertion that α is a natural transformation.

Exactly the same characterisation works for *Bush*. In particular, the arrow $\mathit{bush}\,f$ satisfies

$$\begin{aligned} &\mathit{bush}\,f\;\mathit{NilB} &&= \mathit{NilB} \\ &\mathit{bush}\,f\;(\mathit{ConsB}\,(x,\mathit{xbs})) &&= \mathit{ConsB}\,(f\,x, \mathit{bush}\,(\mathit{bush}\,f)\,\mathit{xbs}) \end{aligned}$$

5 Examples

To illustrate the use of folds over *Nest* and *Bush*, define $\tau :: Q \rightarrow \mathit{List}$ by

$$\tau(x,y) = [x,y]$$

Using τ and the natural transformation $\mathit{concat} :: \mathit{List} \cdot \mathit{List} \rightarrow \mathit{List}$, we have $\mathit{concat} \cdot \mathit{list}\,\tau :: \mathit{List} \cdot Q \rightarrow \mathit{List}$, and so

$$\alpha_{\mathit{List}} \cdot \mathcal{F}(\mathit{concat} \cdot \mathit{list}\,\tau) :: \mathcal{F}(\mathit{List}) \rightarrow \mathit{List}$$

where $\mathcal{F}(F) = K1 + Id \times FQ$ is the higher-order functor associated with *Nest*. The function *listify*, defined by

$$listify = fold\,(\alpha_{List} \cdot \mathcal{F}(concat \cdot list\,\tau))$$

therefore has type $listify :: Nest \to List$. For example, *listify* takes

$$[0,\ (1,1),\ ((2,2),(3,3))] \quad \text{to} \quad [0,1,1,2,2,3,3]$$

The converse function $nestify :: List \to Nest$ can be defined by

$$nestify = fold\,(\alpha_{Nest} \cdot \mathcal{F}(nest\,\delta))$$

where $\mathcal{F}(F) = K1 + Id \times F$ is the higher-order functor associated with *List*, and $\delta\,a = (a, a)$ has type $\delta :: Id \to Q$. For example, *nestify* takes

$$[0,1,2] \quad \text{to} \quad [0,\ (1,1),\ ((2,2),(2,2))]$$

For another example, define $\sigma :: Q \to Bush$ by

$$\sigma(x, y) = [x, [y]]$$

Then $bush\,\sigma :: Bush \cdot Q \to Bush \cdot Bush$, and so

$$\alpha_{Bush} \cdot \mathcal{F}(bush\,\sigma) :: \mathcal{F}(Bush) \to Bush$$

where $\mathcal{F}(F) = K1 + Id \times FQ$ is the functor associated with *Nest*. Hence

$$bushify = fold\,(\alpha_{Bush} \cdot \mathcal{F}(bush\,\sigma))$$

has type $bushify :: Nest \to Bush$. For example, *bushify* sends

$$[1,\ (2,3),\ ((4,5),(6,7))] \quad \text{to} \quad [1,\ [2,[3]],\ [[4,[5]],[[6,[7]]]]]$$

6 The problem

The basic problem with the higher-order approach described above concerns expressive power. Part of the problem is that it does not generalise the standard semantics for regular datatypes; in particular, it does not enable us to make use of the standard instances of *fold* over such datatypes. To see why not, let us compare the two semantics for the datatype *List*.

Under the standard semantics, $fold\,f :: List\;a \to b$ when $f :: 1 + a \times b \to b$. For example,

$$fold\,(zero, plus) :: List\;Int \to Int$$

sums a list of integers, where $zero :: 1 \to Int$ is a constant delivering 0, and $plus :: Int \times Int \to Int$ is binary addition.

As another example,

$$fold\,(nil, cat) :: List\,(List\;a) \to List\;a$$

concatenates a list of lists; this function was called *concat* above. The binary operator *cat* has type $cat :: List\ a \times List\ a \to List\ a$ and concatenates two lists.

Under the new semantics, $fold\,\varphi :: List \to F$ when $\varphi :: K1 + Id \times F \to F$. We can no longer sum a list of integers with such a fold because *plus* is not a natural transformation of the right type. For $fold\,(zero, plus)$ to be well-typed we require that *plus* has type $plus :: Id \times KInt \to KInt$. Thus,

$$plus_a :: a \times Int \to Int$$

for all a, and so *plus* would have to ignore its first argument.

Even worse, we cannot define $concat :: List \cdot List \to List$ as an instance of *fold*, even though it is a natural transformation. The binary concatenation operator *cat* does not have type

$$cat :: Id \times List \to List$$

because again it would have to ignore its first argument. Hence $fold\,(nil, cat)$ is not well-typed.

On the other hand, $\alpha_{Nest} \cdot \mathcal{F}(nest\,\delta)$ does have type $K1 + Id \times Nest \to Nest$, so the definition of *nestify* given in the previous section is legitimate.

Putting the problem another way, in the standard semantics, $fold\,f$ is defined by providing an arrow $f :: F(a, b) \to b$ for a fixed a and b; we cannot in general elevate f to a natural transformation that is parametric in a.

7 An alternative

Fortunately, for lists and other regular datatypes, there is a way out of this particular difficulty. Using the isomorphism defining *List*, the functor $List \cdot F$ satisfies the isomorphism

$$List \cdot F \cong (K1 + Id \times List) \cdot F \cong K1 + F \times (List \cdot F)$$

Hence $List \cdot F$ is isomorphic to the "higher-order" datatype $Listr\ F$, declared by

$$\textbf{data}\ Listr\ F \xleftarrow{\alpha} K1 + F \times Listr\ F$$

We can write the functor on the right as $\mathcal{F}(F, Listr\ F)$, where $\mathcal{F}$ now is a higher-order binary functor of type

$$Nat(\mathbf{C}) \times Nat(\mathbf{C}) \to Nat(\mathbf{C})$$

Over the higher-order datatype $Listr\ F$, the natural transformation $fold\,\varphi$ takes an arrow $\varphi :: K1 + F \times G \to G$, and has type $fold\,\varphi :: Listr\ F \to G$. If we change $Listr\ F$ to $List \cdot F$ in this signature, we have a useful fold operator for lists. In particular,

$$fold\,(zero, plus) :: List \cdot KInt \to KInt$$

since $(zero, plus) :: K1 + KInt \times KInt \to KInt$. The arrow $fold\,(zero, plus)$ of $Nat(\mathbf{C})$ is a natural transformation; since $List \cdot KInt = K(List\ Int)$, its component for any a is the standard fold $fold\,(zero, plus) :: List\ Int \to Int$.

By a similar device, all folds in the standard semantics are definable as folds in the new semantics, simply by lifting the associated algebra to be a natural transformation between constant functors.

More precisely, define $\mathit{Type}\,a$ to be the least fixed point of a regular functor F_a, where $F_a(b) = F(a, b)$. Furthermore, define $\mathit{Typer}\,G$ to be the least fixed point of $\mathcal{F}_G$, where $\mathcal{F}_G(H) = \mathcal{F}(G, H)$ and $\mathcal{F}(G, H)x = F(Gx, Hx)$ for all objects x. Take an algebra $f :: F(a, b) \to b$, and construct the natural transformation $\varphi :: \mathcal{F}(Ka, Kb) \to Kb$ by setting $\varphi = Kf$. This is type correct since

$$\mathcal{F}(Ka, Kb)x = F(Ka(x), Kb(x)) = F(a, b) \quad \text{and} \quad Kb(x) = b$$

Then $\mathit{fold}\,f :: \mathit{Type}\,a \to b$, and $\mathit{fold}\,\varphi :: \mathit{Typer}\,Ka \to Kb$ satisfy

$$\mathit{fold}\,\varphi = K(\mathit{fold}\,f)$$

under the isomorphism $\mathit{Typer}\,Ka = K(\mathit{Type}\,a)$.

Thus, not only do we generalise from the defining expression for *List* by replacing occurrences of *List* by G, we also generalise by replacing occurrences of *Id* by a functor F.

However, the same idea does *not* work for nested datatypes such as *Nest*. This time we have

$$\mathit{Nest} \cdot F \cong (K1 + \mathit{Id} \times (\mathit{Nest} \cdot Q)) \cdot F \cong K1 + F \times (\mathit{Nest} \cdot Q \cdot F)$$

The type $\mathit{Nest} \cdot F$ is quite different from the datatype defined by

$$\textbf{data}\ \mathit{Nestr}\,F \xleftarrow{\alpha} K1 + F \times ((\mathit{Nestr}\,F) \cdot Q)$$

For example, $\mathit{Nest}\,(\mathit{List}\,a)$ is the type of nests of lists over a, so the n-th entry of such a nest has type $Q^n\,(\mathit{List}\,a)$. On the other hand the n-th entry of a nest of type $\mathit{Nestr}\,\mathit{List}\,a$ has type $\mathit{List}\,(Q^n\,a)$.

Even more dramatically, the type $\mathit{Nest}\,\mathit{Int}$ gives a nest of integers, but $\mathit{Nestr}\,\mathit{KInt}\,b$ is isomorphic to ordinary lists of integers for all b. More generally, $\mathit{Nestr}\,Ka$ is the constant functor $K(\mathit{List}\,a)$.

On the other hand, we have $\mathit{Nest} = \mathit{Nestr}\,\mathit{Id}$, so the higher-order view is indeed a generalisation of the previous one.

8 Reductions

Replacing higher-order unary functors by higher-order binary functors enables us to integrate the standard theory of regular datatypes into the proposed scheme. Unfortunately, while the higher-order approach is elegant and generic, it seems limited in the scope of its applicability to nested datatypes, which is restricted to folding with natural transformations. For example, one cannot sum a nest of integers with a fold over nests. Such a computation is an instance of a useful general pattern called a *reduction*. It is possible to define reductions completely generically for all regular types (see [15]), but we do not know at present whether the same can be done for nested datatypes.

One way to sum a nest of integers is by first listifying the nest and then summing the result with a fold over lists. More generally, this strategy can be used to reduce a nest with an arbitrary binary operator $\oplus$ and a seed e. For example,

$$[x_0,\ (x_1, x_2),\ ((x_3, x_4), (x_5, x_6))]$$

reduces to

$$x_0 \oplus (x_1 \oplus (x_2 \oplus \cdots \oplus (x_6 \oplus e)))$$

It can be argued that this strategy for reducing over nests is unsatisfactory because the structure of the nest entries is not reflected in the way in which $\oplus$ is applied. Better is to introduce a second operator $\otimes$ and reduce the nest above to

$$x_0 \otimes ((x_1 \oplus x_2) \otimes (((x_3 \oplus x_4) \oplus (x_5 \oplus x_6)) \otimes e))$$

By taking $\otimes$ to be $\oplus$, we obtain another way of reducing a nest.

The above pattern of computation can be factored as a fold over lists after a reduction to a list:

$$\mathit{fold}\,(e, \otimes) \cdot \mathit{reduce}\,(\oplus)$$

With $(\oplus) :: Q\ a \to a$, the function $\mathit{reduce}\,(\oplus)$ has type $\mathit{Nest}\ a \to \mathit{List}\ a$. For example, applied to the nest above, $\mathit{reduce}\,(\oplus)$ produces

$$[x_0,\ x_1 \oplus x_2,\ (x_3 \oplus x_4) \oplus (x_5 \oplus x_6)]$$

There is no problem with defining *reduce*. In a functional style we can define

$$\begin{aligned}
&\mathit{reduce}\ \mathit{op}\ \mathit{NilN} &&= \mathit{NilL} \\
&\mathit{reduce}\ \mathit{op}\ (\mathit{ConsN}\ (x, \mathit{xps})) &&= \mathit{ConsL}\,(x, \mathit{reduce}\ \mathit{op}\ (\mathit{nest}\ \mathit{op}\ \mathit{xps}))
\end{aligned}$$

In effect, *reduce op* applies the following sequence of functions to the corresponding entries of a nest:

$$[\mathit{id},\ \mathit{op},\ \mathit{op} \cdot \mathit{square}\ \mathit{op},\ \mathit{op} \cdot \mathit{square}\ \mathit{op} \cdot \mathit{square}\,(\mathit{square}\ \mathit{op}), \ldots]$$

The n-th element of this sequence has type $Q^n\ a \to a$ when $\mathit{op} :: Q\ a \to a$.

The reduction of a bush proceeds differently:

$$\begin{aligned}
&\mathit{reduce}\,(e, \mathit{op})\ \mathit{NilB} = e \\
&\mathit{reduce}\,(e, \mathit{op})\ (\mathit{ConsB}\ (x, \mathit{xbs})) = \\
&\qquad\qquad \mathit{op}\ (x, \mathit{reduce}\,(e, \mathit{op})\ (\mathit{bush}\,(\mathit{reduce}\,(e, \mathit{op}))\ \mathit{xbs}))
\end{aligned}$$

At present we see no systematic way of unifying reductions over nested datatypes, nor of relating them to the folds of previous sections.

9 Another approach

There is a way that higher-order folds and the reductions of the previous section can be unified, but whether or not the method is desirable from a calculational point of view remains to be seen. It requires a different and more complicated notion of folding over a nested datatpe, one that involves an *infinite* sequence of appropriate algebras to replace the infinite sequence of differently typed instances of the constructors of the datatype. We will briefly sketch the construction for the type $Nest\ a$.

The basic idea is to provide an infinite sequence of algebras to replace the constructor $\alpha = (NilN, ConsN)$ of $Nest$, one for each instance

$$\alpha :: F(Q^n\ a, Nest\ (Q^{n+1}\ a)) \to Nest\ (Q^n\ a)$$

where n is a natural number and $F(a, b) = 1 + a \times b$. For regular datatypes the application of $fold\, f$ to a term can be viewed as the systematic replacement of the constructors by corresponding components of f, followed by an evaluation of the result. The same idea is adopted here for nested datatypes. However, whereas for regular datatypes each occurrence of a constructor in a term has the same typing, the same is not true for nested datatypes, hence the need to provide a collection of replacements.

In more detail, consider the datatype $NestAlgs$ defined by

$$\mathbf{data}\ NestAlgs\ G\ (a, b) = Cons\ (F(a, G(Qb)) \to Gb,\ NestAlgs\ G\ (Qa, Qb))$$

The datatype $NestAlgs$ is a *coinductive*, infinite, nested datatype. The n-th entry of a value of type $NestAlgs\ G\ (a, b)$ is an algebra of type

$$F(Q^n\ a, G(Q^{n+1}\ b)) \to G(Q^n\ b)$$

Now for $fs :: NestAlgs\ G\ (a, b)$, define $fold\, fs :: Nest\ a \to Gb$ by the equation

$$fold\, fs \cdot \alpha = head\, fs \cdot F(id, fold\ (tail\, fs))$$

where

$$\begin{aligned} head\ (Cons\ (f, fs)) &= f \\ tail\ (Cons\ (f, fs)) &= fs \end{aligned}$$

Equivalently,

$$fold\ (Cons\ (f, fs)) \cdot \alpha = f \cdot F(id, fold\, fs)$$

To illustrate this style of $fold$, suppose $f :: a \to b$ and define $generate\, f :: NestAlgs\ Nest\ (a, b)$ by

$$generate\, f = Cons\ (\alpha \cdot F(f, id), generate\ (square\, f))$$

Then $fold\ (generate\, f) :: Nest\ a \to Nest\, b$, and in fact

$$nest\, f = fold\ (generate\, f)$$

The functorial action of *Nest* on arrows can therefore be recovered as a fold. The proof of $nest\,(f \cdot g) = nest\,f \cdot nest\,g$ makes use of coinduction.

As another example, suppose $\varphi :: \mathcal{F}(Id, GQ) \to G$ is a natural transformation, where $\mathcal{F}(M, N)a = F(Ma, Na)$. Define $repeat\,\varphi :: NestAlgs\ G$ by

$$repeat\,\varphi = Cons\,(\varphi, repeat\,\varphi Q)$$

For each type a we have $(repeat\,\varphi)_a :: NestAlgs\ G\,(a,a)$. The relationship between the higher-order folds of the previous sections and the current style of folds is that

$$fold\,\varphi = fold\,(repeat\,\varphi)$$

In particular, $fold\,(repeat\,\alpha) = id :: Nest \to Nest$.

We can also define reductions as an instance of the new folds. Suppose $f :: F(a,a) \to a$, so $f = (f_0, f_1)$, where $f_1 :: Qa \to a$. Define $redalgs\,f :: NestAlgs\ Ka\,(a,b)$ by

$$\begin{aligned} &redalgs\,f = red\ id \\ &\qquad\qquad \textbf{where}\ red\ k \ = \ Cons\,(f \cdot F(k, id), red\,(f_1 \cdot square\ k)) \end{aligned}$$

We have $fold\,(redalgs\,f) :: Nest\ a \to a$, and we claim that

$$reduce\,f = fold\,(redalgs\,f)$$

10 Conclusions

The results of this investigation into nested datatypes are still incomplete and in several aspects unsatisfactory. The higher-order folds are attractive, and the corresponding calculational theory is familiar, but they seem to lack sufficient expressive power. The approach sketched in the previous section for *Nest* is more general, but brings in more machinery. Furthermore, it is not clear what the right extension is to other nested datatypes such as *Bush*.

We have also ignored one crucial question in the foregoing discussion, namely, what is the guarantee that functors such as *Nest* and *Nestr* do in fact exist as least fixed points of their defining equations? The categorical incantation ensuring the existence of an initial F-algebra in a co-complete category $\mathbf{C}$ is that, provided F is co-continuous, it is the colimit of the chain

$$0 \hookrightarrow F0 \hookrightarrow FF0 \hookrightarrow \cdots$$

The category **Fun** has everything needed to make this incantation work: **Fun** is co-complete (in fact, bi-complete) and all regular functors F on **Fun** are co-continuous. The proof for polynomial functors can be found in [14], and the extension to type functors is in [13].

Moreover, the category $Nat\,(\mathbf{Fun})$ inherits co-completeness from the base category **Fun** (see [11, 7]). We believe that all regular higher-order functors are co-continuous, though we have not yet found a proof of this in the literature, so the existence of datatypes like *Nest* and *Bush* is not likely to be problematic.

If we adopt the higher-order approach, then there is a need to give a systematic account of reductions over a nested datatype. If the alternative method of the previous section proves more useful, then there is a need to give a systematic account of the method, not only for an arbitrary inductive nested datatype, but also for coinductive nested datatypes.

Finally, in [4] (see also [5]) the idea was proposed that a datatype was a certain kind of functor called a *relator*, together with a membership relation. It needs to be seen how the notion of membership can be extended to nested datatypes

Acknowledgements

The authors would like to thank Ian Bayley, Jeremy Gibbons, Oege de Moor, Mark Jones, and Simon Peyton Jones for comments and discussions on the work. A particular debt is owed to Ross Paterson, who commented on an earlier draft of the paper. Thanks are also due to the anonymous referees who suggested numerous improvements.

References

1. R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computing Science. Prentice Hall, 1996.
2. R. S. Bird, P. F. Hoogendijk, and O. De Moor. Generic programming with relations and functors. *Journal of Functional Programming*, 6(1):1–28, 1996.
3. R.H. Connelly and F. Lockwood Morris. A generalisation of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, 1995.
4. Oege de Moor and Paul Hoogendijk. What is a datatype? Technical Report 96/16, Department of Maths and Computing Science, Eindhoven University of Technology, 1996.
5. Paul Hoogendijk. *A Generic theory of Data Types*. Ph.D Thesis, Eindhoven University of Technology, 1997.
6. L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1996.
7. Peter Freyd. Algebraically complete categories. Springer-Verlag Lecture Notes in Mathematics, vol 1488, 95–104, 1990.
8. 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.
9. J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Functional Programming and Computer Architecture*, pages 238–248. Association for Computing Machinery, 1995.
10. J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.
11. Saunders Mac Lane. Categories for the Working Mathematician. Graduate Texts in Mathematics. Springer-Verlag, 1971.
12. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

13. G. Malcolm. Algebraic Data Types and Program Transformation. Ph.D thesis, University of Groningen, The Netherlands, 1990.
14. E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computing Science. Springer-Verlag, 1986.
15. Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations Logics, and Programs Proceedings Eighth International Symposium PLILP '96*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.
16. E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Functional Progamming Languages and Computer Architecture*, pages 324–333. Association for Computing Machinery, 1995.
17. A. Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume LNCS 167, pages 217–228. Springer-Verlag, 1984.
18. C. Okasaki. *Purely Functional Data Structures*. Ph.D thesis, School of Computer Science, Carnegie Mellon University, 1996.
19. C. Okasaki. Catenable double-ended queues. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 66–74. ACM, 1997.
20. S. Peyton Jones and J. Launchbury. Explicit quantification in Haskell. See: `http://www.dcs.gla.ac.uk/people/personal/simonpj/`.