On the Semantics of Nested Datatypes

Clare Martin Jeremy Gibbons

February 7, 2001

Keywords

Data structures; first-order and higher-order nested datatypes; formal semantics; functional programming; functor categories; program derivation; programming calculi.

1 Introduction

Nested datatypes were introduced in [1] as recursively defined parameterised datatypes in which the parameter of the datatype changes in the recursive call. For example, the *Nest* datatype, representing perfectly balanced binary trees, can be defined in the programming language Haskell [11] by

$$\textbf{data}\ \mathit{Nest}\, a = \mathit{Nil} \mid \mathit{Cons}\,(a, \mathit{Nest}\,(\mathit{Pair}\, a)) \tag{1.1}$$

where $\mathit{Pair}\, a = a \times a$. In contrast, for *regular* datatypes the parameter is unchanged:

$$\textbf{data}\ \mathit{Tree}\, a = \mathit{Empty} \mid \mathit{Node}\,(a, \mathit{Pair}\,(\mathit{Tree}\, a))$$

Regular datatypes can be given a semantics as initial algebras in the category **Set** of sets and total functions [7], but it was shown in [1] that this theory is inadequate to describe nested datatypes. Instead, one solution proposed there was to define them as initial algebras in the functor category $\mathbf{Nat}(\mathbf{Set})$, where $\mathbf{Nat}(\mathbf{C})$ is the category with objects all endofunctors on $\mathbf{C}$ and arrows all natural transformations between them. It was observed, however, that this approach had not been validated, in the sense of ensuring that such initial algebras always exist. Blampied's recent thesis [3] has only a weak result about existence in certain special cases. We show here that initial algebras are not guaranteed to exist in the functor category itself, but that they do exist in some other categories, and in particular the subcategory of all *cocontinuous* endofunctors and natural transformations between them. This category is then a semantic domain for nested datatypes, both first order as in [1, 2] and higher-order as in [4, 5].

2 Regular Datatypes

We will assume throughout that $\mathbf{C}$ is an ω-cocomplete category with initial object and all finite products and coproducts, where ω is the poset of naturals taken as a category. Then a polynomial functor $\mathbf{C} \to \mathbf{C}$ is any functor which can be constructed from constant functors and the identity functor, through the use of the composition, product or coproduct operators, as listed below in decreasing order of precedence.

(i) Id, the identity functor
(ii) K_a, the constant functor for each object a
(iii) $F \cdot G$, the composition of F after G
(iv) $F \dot{\times} G$, the product of F and G
(v) $F \dot{+} G$, the coproduct of F and G

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

$$\begin{array}{lcl} (F \mathbin{\dot{\oplus}} G)\, c & = & F\, c \oplus G\, c \\ (F \mathbin{\dot{\oplus}} G)\, f & = & F\, f \oplus G\, f \\ (\theta \mathbin{\dot{\oplus}} \psi)\, c & = & \theta\, c \oplus \psi\, c \end{array} \tag{2.1}$$

for objects c and arrows f of $\mathbf{C}$.

By definition, the class of *regular* datatypes is the closure under least fixed points of the class of polynomial functors shown above. For example, one of the simplest examples of a regular datatype is the type $\mathit{List}\, a$ of all lists with elements of some type a. The polynomial functor L_a that is used to define this datatype is

$$L_a = K_1 \mathbin{\dot{+}} K_a \mathbin{\dot{\times}} \mathit{Id}$$

where 1 denotes the terminal object. Since $\mathit{List}\, a$ is a fixed point of this functor, we have the familiar recursive pattern:

$$\mathit{List}\, a \cong L_a\, (\mathit{List}\, a) = 1 + a \times \mathit{List}\, a$$

3 Nested Datatypes

The class of functors used to define nested datatypes in [2] is the closure under least fixed points of a larger class of functors: the class of functors above, together with one extra operator that we will now define. This operator can only be defined in the functor category $\mathbf{Nat(C)}$, since it is defined using the horizontal composition bifunctor $\star : \mathbf{Nat(C)} \times \mathbf{Nat(C)} \to \mathbf{Nat(C)}$, which maps each pair of functors $P, Q : \mathbf{C} \to \mathbf{C}$ to its ordinary composite:

$$P \star Q = P \cdot Q$$

and each pair of natural transformations $\theta : P \to R$ and $\psi : Q \to S$ to the horizontal composite $\theta \star \psi : P \star Q \to R \star S$ defined by

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

Nested datatypes are defined in terms of higher-order functors or *hofunctors*, which is the name given to endofunctors on $\mathbf{Nat(C)}$ in [2]. *Polynomial hofunctors* are constructed from the identity and constant hofunctors $\mathcal{I}d$ and $\mathcal{K}_F$, composition $\cdot$, and the twice-lifted (co-)products $\ddot{\times}$ and $\ddot{+}$, together with

(vi) $\mathcal{F} \mathbin{\dot{\star}} \mathcal{G}$, the horizontal composition of $\mathcal{F}$ and $\mathcal{G}$.

As an example of the use of this operator, consider the *Nest* datatype (1.1). It was claimed in [1] that this datatype can be defined as the initial algebra of the hofunctor $\mathcal{N} : \mathbf{Nat(C)} \to \mathbf{Nat(C)}$ defined for each functor $F : \mathbf{C} \to \mathbf{C}$ and natural transformation $\eta : F \to G$ by

$$\begin{array}{lcl} \mathcal{N}(F) & = & K_1 \mathbin{\dot{+}} \mathit{Id} \mathbin{\dot{\times}} F \cdot \mathit{Pair} \\ \mathcal{N}(\eta) & = & \mathit{id}K_1 \mathbin{\dot{+}} \mathit{id} \mathbin{\dot{\times}} \eta \mathit{Pair} \end{array}$$

The definition of $\mathcal{N}$ can be written more succinctly using horizontal composition:

$$\mathcal{N} = \mathcal{K}_{K_1} \mathbin{\ddot{+}} \mathcal{K}_{Id} \mathbin{\ddot{\times}} (\mathcal{I}d \mathbin{\dot{\star}} \mathcal{K}_{Pair}) \tag{3.1}$$

If *Nest* is an initial algebra of $\mathcal{N}$, then it must satisfy $\mathcal{N}\,(\mathit{Nest}) \cong \mathit{Nest}$, since the initial algebra of any functor is a fixed point of that functor. This equation corresponds to the recursive definition (1.1). So demonstrating that datatype definitions like (1.1) make sense amounts to showing that initial algebras of hofunctors like $\mathcal{N}$ exist.

4 Cocontinuity

Suppose that $\mathbf{C}$ is a cocomplete category with initial object $\mathbf{0}$. If $F : \mathbf{C} \to \mathbf{C}$ is an ω-cocontinuous functor, then its initial algebra exists and is given by the colimit of the chain

$$\mathbf{0} \to F\mathbf{0} \to F(F\mathbf{0}) \to \cdots$$

The cocontinuity properties of the constructors of regular datatypes were described in [8]. It was shown there that the constant and identity functors are ω-cocontinuous in any category $\mathbf{C}$, as is the composition of any pair of ω-cocontinuous functors. If $\mathbf{C}$ has coproducts, then the coproduct of two ω-cocontinuous functors is ω-cocontinuous. In some categories, such as $\mathbf{Set}$, the product of two ω-cocontinuous functors is also ω-cocontinuous. Since the pointwise extension of bifunctors to the functor category given in (2.1) preserves colimits ([6], V.3), all of the constructors of polynomial datatypes also preserve ω-cocontinuity in $\mathbf{Nat}(\mathbf{Set})$. It remains therefore to determine whether the horizontal composition $F \star G$ of two ω-cocontinuous functors F and G is ω-cocontinuous. We will begin with a formal definition of ω-cocontinuity.

Definition 4.1 (right chain) In any category $\mathbf{C}$, a *right chain* is a functor $P : \omega \to \mathbf{C}$, or equivalently, a collection of arrows $\{p_n : P\,n \to P\,(n{+}1) \mid n \geq 0\}$. For such a collection we define ${}_jp_i = p_{j-1} \cdot p_{j-2} \cdot \ldots \cdot p_i : P\,i \to P\,j$ for $i \leq j$, and note that ${}_kp_j \cdot {}_jp_i = {}_kp_i$ when $i \leq j \leq k$.

Definition 4.2 (ω-cocone) The pair (T, τ) is a *ω-cocone* of a chain P if τ is a natural transformation from P to K_T, or equivalently, if τ is a family of functions $\tau_n : P\,n \to T$ such that for all $n \geq 0$,

$$\tau_{n+1} \cdot p_n = \tau_n \tag{4.3}$$

Definition 4.4 (ω-colimit) An *ω-colimit* of a right chain P is a cocone (A, α) that is universal among cocones. Equivalently, for all cocones (T, τ) of P there is an arrow $\tau/\alpha : A \to T$ such that for all $f : A \to T$,

$$f = \tau/\alpha \quad \equiv \quad (\forall n \geq 0 : \tau_n = f \cdot \alpha_n) \tag{4.5}$$

The notation τ/α comes from [7]. If (A, α) is the ω-colimit of P, then we will usually write Colim P for the object A, and will sometimes add subscripts for clarity and write $\mathrm{Colim}_i\,(P\,i)$.

Example 4.6 (ω-colimits in Set) It is shown in [8] that every right chain P in $\mathbf{Set}$ has an ω-colimit defined as follows. Let $U = \{(n, x) \mid x \in P\,n, n \geq 0\}$ and define an equivalence relation $\sim$ on U by

$$(n, x) \sim (m, y) \quad \text{if } \exists k \geq m, n \text{ such that } {}_kp_n(x) = {}_kp_m(y) \text{ in } P\,k$$

Then Colim $P = A$ where A is the set of $\sim$-equivalence classes of U, and the cocone arrows $\alpha_n : P\,n \to A$ are defined by $\alpha_n(x) = [n, x]$, where $[n, x]$ is the $\sim$-equivalence class of (n, x).

Definition 4.7 (ω-cocontinuous) A functor $F : \mathbf{C} \to \mathbf{D}$ is *ω-cocontinuous* if for all $P : \omega \to \mathbf{C}$

$$F(\mathrm{Colim}\ P) \cong \mathrm{Colim}(F \cdot P)$$

It follows from this that the arrows of the corresponding cocones of $F \cdot P$ are isomorphic to each other.

The following example of a functor that is not ω-cocontinuous is from [8].

Example 4.8 (non-ω-cocontinuous functor) Let $\mathbb{P} : \mathbf{Set} \to \mathbf{Set}$ be the functor that maps each set X to its powerset $\mathbb{P}X$, and each function $f : X \to Y$ to the function $\mathbb{P}f$ given by

$$\mathbb{P}f\,S = \{f\,x \mid x \in S\}$$

Let $P : \omega \to \mathbf{Set}$ map each natural number n to the set of natural numbers from 0 to n, denoted by $\{0..n\}$, and each arrow of ω to an inclusion. This gives the right chain $\{0\} \subset \{0, 1\} \subset \{0, 1, 2\} \subset \cdots$. It is clear from Example 4.6 that the colimit of this chain is $(\mathbb{N}, \alpha)$ where $\alpha_n(x) = x$. If we apply the functor $\mathbb{P}$ to this chain we obtain the chain $\mathbb{P}\{0\} \subset \mathbb{P}\{0, 1\} \subset \mathbb{P}\{0, 1, 2\} \subset \cdots$. Again, by Example 4.6, we see that the colimit of this chain is obtained by partitioning a countable union of finite sets into equivalence classes, and is therefore countable. The set $\mathbb{P}\mathbb{N}$ is not countable, however, and is therefore not the colimit of this chain.

5 The Problem

We will show that the horizontal composition bifunctor $\star$ is not ω-cocontinuous in $\mathbf{Nat}(\mathbf{Set})$, by constructing a right chain based on Example 4.8 whose colimit is not preserved by $\star$. Let $P : \omega \to \mathbf{Nat}(\mathbf{Set}) \times \mathbf{Nat}(\mathbf{Set})$ map each natural number n to

$$P\,n = (\mathbb{P}, K_{\{1..n\}} \dot{+} \mathbb{P})$$

and each arrow of ω to the obvious inclusion. Since colimits are defined pointwise in $\mathbf{Nat}(\mathbf{Set})$, and product and coproduct are ω-cocontinuous, we have

$$\text{Colim } P = (\mathbb{P}, K_{\mathbb{N}^+} \dot{+} \mathbb{P}) \tag{5.1}$$

where $\mathbb{N}^+$ is the set of all positive natural numbers, and the colimit arrows are inclusions. We will show that P is not ω-cocontinuous by applying the relevant functors to the empty set $\emptyset$. For convenience we will denote the singleton set $\mathbb{P}\emptyset$ by 0, and therefore write the set $\{1..n\} + \mathbb{P}\emptyset$ as $\{0..n\}$. Then, using the Haskell convention of writing the prefix form of the infix operator $\star$ as $(\star)$, we have

$$((\star)\,(\text{Colim } P))\,\emptyset = (\mathbb{P} \star (K_{\mathbb{N}^+} \dot{+} \mathbb{P}))\,\emptyset \cong \mathbb{P}\mathbb{N}$$

which is not countable, but

$$\text{Colim}\,((\star)P)\,\emptyset = \text{Colim}_n\,((\star)(P\,n))\,\emptyset = \text{Colim}_n\,(\mathbb{P} \star (K_{1..n} \dot{+} \mathbb{P}))\,\emptyset \cong \text{Colim}_n\,\mathbb{P}\{0..n\}$$

which is countable by the same argument used in Example 4.8. Therefore $\star$ is not ω-cocontinuous.

6 The Solution

Since horizontal composition is not ω-cocontinuous in $\mathbf{Nat}(\mathbf{Set})$, we need to determine a subcategory in which it is ω-cocontinuous, and hence find a more suitable semantic category for nested datatypes. We will begin by giving some definitions and results from [6] which show that a bifunctor $F : \mathbf{A} \times \mathbf{B} \to \mathbf{C}$ is ω-cocontinuous if it is ω-cocontinuous in each argument separately.

Definition 6.1 (comma category) If $L : \mathbf{J}' \to \mathbf{J}$ is a functor and k is an object of $\mathbf{J}$, then the *comma category* $(k \downarrow L)$ has as objects all pairs (f, j) where j is an object of $\mathbf{J}'$ and $f : k \to L\,j$, and as arrows $h : (f, j) \to (f', j')$ all arrows $h : j \to j'$ in $\mathbf{J}'$ for which $f' = L\,h \cdot f$.

Definition 6.2 (final functor) A functor $L : \mathbf{J}' \to \mathbf{J}$ is *final* if for each object k in $\mathbf{J}$, the comma category $(k \downarrow L)$ is non-empty and connected.

In particular, the diagonal functor $\Delta : \omega \to \omega \times \omega$ defined by $\Delta\,i = (i, i)$ is final, since for each $j, k \in \mathbb{N}$ the comma category $(j, k) \downarrow \Delta$ is the set of objects $\{((j,k) \mapsto (l,l), l) \mid j \le l \text{ and } k \le l\}$, where $(j, k) \mapsto (l, l)$ stands for the arrow from (j, k) to (l, l) in $\omega \times \omega$. Each pair of objects is then connected by a unique arrow in $\omega \times \omega$.

The propositions below follow from results in Sections IX.3 and IX.8 of [6] respectively. These propositions use a more general notion of colimit than that defined in Section 4; however, specialised to right chains, that more general notion coincides with our (4.4), which is sufficient for the proof of Theorem 6.5 below to carry through.

Proposition 6.3 If $\mathbf{C}$ is cocomplete, $L : \mathbf{J}' \to \mathbf{J}$ is final and $F : \mathbf{J} \to \mathbf{C}$ is a functor, then $\text{Colim}\,(F \cdot L)$ is isomorphic to $\text{Colim } F$.

Proposition 6.4 If $\mathbf{C}$ is cocomplete, and $F : \mathbf{I} \times \mathbf{J} \to \mathbf{C}$, where $\mathbf{I}$ and $\mathbf{J}$ are small, then

$$\text{Colim}_i\ \text{Colim}_j\ F(i, j) \cong \text{Colim}_{(i,j)}\ F(i, j) \cong \text{Colim}_j\ \text{Colim}_i\ F(i, j)$$

We will now use these two propositions to show that a bifunctor is ω-cocontinuous if it is ω-cocontinuous in each argument separately. We will use the notation $\vartriangle$ to denote the *split* of two arrows: if $F : \mathbf{A} \to \mathbf{B}$ and $G : \mathbf{A} \to \mathbf{C}$, then their split is the standard product arrow $F \vartriangle G : \mathbf{A} \to \mathbf{B} \times \mathbf{C}$.

Theorem 6.5 If $\mathbf{C}$ is cocomplete, and $F : \mathbf{A} \times \mathbf{B} \to \mathbf{C}$ is a bifunctor with the property that for any $a \in \mathbf{A}$, $b \in \mathbf{B}$ the functors

$$F_a = F \cdot (K_a \vartriangle Id) \tag{6.6}$$

$$F_b = F \cdot (Id \vartriangle K_b) \tag{6.7}$$

are ω-cocontinuous, then F is ω-cocontinuous.

Proof Suppose $P : \omega \to \mathbf{A} \times \mathbf{B}$, and let $R : \omega \to \mathbf{A}$ and $S : \omega \to \mathbf{B}$ be defined by

$$R = Fst \cdot P \text{ and } S = Snd \cdot P \tag{6.8}$$

where *Fst* and *Snd* denote the projections from the product. Then

$$\begin{array}{ll}
 & \mathrm{Colim}\,(F \cdot P) \\
= & \quad \text{property of } \times \\
 & \mathrm{Colim}\,(F \cdot R \times S \cdot \Delta) \\
\cong & \quad \text{Proposition 6.3} \\
 & \mathrm{Colim}\,(F \cdot R \times S) \\
\cong & \quad \text{Proposition 6.4} \\
 & \mathrm{Colim}_i\ \mathrm{Colim}_j\,(F(R\,i, S\,j)) \\
= & \quad (6.6) \\
 & \mathrm{Colim}_i\,(F(R\,i, \mathrm{Colim}\ S)) \\
= & \quad (6.7) \\
 & F(\mathrm{Colim}\ R, \mathrm{Colim}\ S) \\
= & \quad \text{cocontinuity of } \times \\
 & F(\mathrm{Colim}\ R \times S) \\
= & \quad \text{Proposition 6.3} \\
 & F(\mathrm{Colim}\,(R \times S \cdot \Delta)) \\
= & \quad \text{property of } \times \\
 & F(\mathrm{Colim}\ P)
\end{array}$$

□

Theorem 6.5 can be used to determine subcategories of $\mathbf{Nat}(\mathbf{C})$ in which horizontal composition is ω-cocontinuous. If we take F to be the horizontal composition bifunctor, then the functor F_b defined in (6.7) is ω-cocontinuous, since colimits are constructed pointwise ([6], V.3). The functor F_a of (6.6) is ω-cocontinuous if $\mathbf{A}$ and $\mathbf{B}$ are taken to be the category of ω-cocontinuous functors and natural transformations between them, which we will call $\mathbf{Coc}(\mathbf{C})$. Its colimits are defined in the same way as those in $\mathbf{Nat}(\mathbf{C})$, since the colimit of a chain of ω-cocontinuous functors is itself ω-cocontinuous. Horizontal composition is also defined in the same way since the composition of two ω-cocontinuous functors is ω-cocontinuous. So this category may be used as a semantic domain for nested datatypes. Since the only assumptions about the category $\mathbf{C}$ were that it was ω-cocomplete with all finite products and coproducts, this result may be applied to categories other than $\mathbf{Set}$. In particular, it can be used to give a semantics for *higher order* nested datatypes, as we will now see.

7 Higher Order Nested Datatypes

A higher order nested datatype is one that is parameterised by type constructors, rather than types. For example, a second order definition in Haskell of the *Nest* datatype is as follows:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{SNest}\ f\ a & = & \mathit{Nil} \mid \mathit{Cons}\,(f\ a, (\mathit{SNest}\,(\mathit{PairF}\ f)\ a)) \\
\textbf{data}\ \mathit{PairF}\ f\ a & = & \mathit{MkPairF}\,(f\ a, f\ a)
\end{array}$$

We could then define an equivalent of the first order *Nest* type (1.1) by:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Leaf}\ a & = & \mathit{MkLeaf}\ a \\ \textbf{type}\ \mathit{Nest}\ a & = & \mathit{SNest}\ \mathit{Leaf}\ a \end{array}$$

One advantage of this second order definition is that it may be easily adapted to other datatypes. It also permits the use of a class-based programming technique that leads to very concise programs [4].

Not all second order nested datatypes have equivalent first order definitions. For example, square matrices can be defined as a second order nested datatype in Haskell by the following [4, 10]:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Cons}\ f\ a & = & \mathit{MkCons}\ a\ (f\ a) \\ \textbf{data}\ \mathit{Nil}\ a & = & \mathit{MkNil} \\ \textbf{data}\ \mathit{SM}\ f\ a & = & \mathit{Zero}\ (f\ (f\ a)) \mid \mathit{Succ}\ (\mathit{SM}\ (\mathit{Cons}\ f)\ a) \\ \textbf{type}\ \mathit{SqMat}\ a & = & \mathit{SM}\ \mathit{Nil}\ a \end{array}$$

but we know of no representation of square matrices as a first order nested datatype. There are many more examples of second order nested datatypes that can be defined in Haskell [4, 5, 10], but so far no-one has given a semantics of these datatypes. The definition of square matrices above could be given a meaning if we could define a hofunctor

$$\mathcal{SM} : \mathbf{Coc}(\mathbf{Set}) \to \mathbf{Coc}(\mathbf{Set})$$

using the following recursive definition for each ω-cocontinuous functor F:

$$\mathcal{SM}(F) \cong (F \star F) \dotplus \mathcal{SM}(Id \dot{\times} F).$$

and a corresponding definition for each natural transformation in $\mathbf{Coc}(\mathbf{Set})$. Then we could define $\mathit{SqMat} = \mathcal{SM}(K_1)$. This kind of definition has a familiar form, since it involves a recursive call which does not have the same arguments on the left-hand side as the right, just like a first order nested datatype. The difference is that it is a definition of a hofunctor, rather than an ordinary functor, so existence of the initial algebra is only guaranteed if the appropriate constructors are shown to reside in $\mathbf{Coc}(\mathbf{Coc}(\mathbf{Set}))$, rather than in $\mathbf{Coc}(\mathbf{Set})$. Fortunately this is the case, since the limits and colimits of $\mathbf{Set}$ are inherited by both these categories. So the (second order) hofunctor $\mathcal{SM}$ is the initial algebra of the (third order) hofunctor $\mathcal{H}$ defined by:

$$\begin{array}{lcl} \mathcal{H} & : & \mathbf{Coc}(\mathbf{Coc}(\mathbf{Set})) \to \mathbf{Coc}(\mathbf{Coc}(\mathbf{Set})) \\ \mathcal{H} & = & \mathcal{K}_{\mathcal{I}d \dot{\star} \mathcal{I}d} \ddot{+} \mathcal{I}d \dot{\star} \mathcal{K}_{\mathcal{K}_{Id} \ddot{\times} \mathcal{I}d} \end{array}$$

(with a corresponding definition for natural transformations), since then for any $\mathcal{P} : \mathbf{Coc}(\mathbf{Set}) \to \mathbf{Coc}(\mathbf{Set})$ we have

$$\mathcal{H}(\mathcal{P}) \quad = \quad (\mathcal{I}d \dot{\star} \mathcal{I}d) \dot{+} \mathcal{P} \cdot (\mathcal{K}_{Id} \ddot{\times} \mathcal{I}d)$$

Since $\mathcal{SM}$ is initial, we have $\mathcal{SM} \cong \mathcal{H}(\mathcal{SM})$, so for any ω-cocontinuous functor $F : \mathbf{Set} \to \mathbf{Set}$, we have

$$\begin{array}{lcl} \mathcal{SM}(F) & \cong & \mathcal{H}(\mathcal{SM})(F) \\ & = & (F \star F) \dotplus \mathcal{SM}\ (Id \dot{\times} F) \end{array}$$

and $\mathit{SqMat} = \mathcal{SM}(K_1)$ is the definition we require.

8 Conclusion

We have shown that the initial algebra semantics of nested datatypes is valid in a wide class of categories. The reason that this is useful is that there is a unique morphism (a *fold* or *catamorphism* [9]) from each initial $\mathcal{F}$-algebra to any other $\mathcal{F}$-algebra $\phi : \mathcal{F}\,F \to F$. Folds are standard operators in functional programming, and satisfy fusion laws that are useful for program transformation. Unfortunately there is a problem with defining them in functor categories, since the parameters to the fold must be natural transformations and therefore polymorphic functions. This means that standard non-polymorphic functions like *sum* and *member* cannot be defined as folds. One solution to this problem is provided by the *generalised folds* of [2]. The results of Section 6 ensure that generalised folds can be defined in any category satisfying the conditions of Theorem 6.5. One such category is the category of relations, so they could also be defined in a relational setting. We have used another such category to provide a semantics for second order nested datatypes; so in theory, generalised folds could also be defined on second order datatypes. The task remains to put this theory into practical use in Haskell; this is a topic of continuing research.

Acknowledgements

We would like to thank Ian Bayley, Richard Bird and Ross Paterson for their helpful comments on the work presented here, and especially Steve Lack who, among other contributions, generalized our original proof of Theorem 6.5. We would also like to thank the anonymous referees, whose thorough reports greatly improved the presentation of this paper.

References

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

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

[3] P. A. Blampied. *Structured recursion for non-uniform data-types.* PhD thesis, University of Nottingham, 2000.

[4] R. Hinze. *Numerical Representations as Higher-Order Nested Datatypes.* Technical Report IAI-TR-98-12, Institüt fur Informatik III, Universität Bonn, 1998.

[5] R. Hinze. Manufacturing datatypes. In *Workshop on Algorithmic Aspects of Advanced Programming Languages*, p. 1–16. Columbia University technical report CUCS-023-99, 1999.

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

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

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

[9] E. Meijer, M. Fokkinga and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *LNCS 523: Functional Programming Langage and Computer Architecture*, p. 124–144, 1991.

[10] C. Okasaki. From fast exponentiation to square matrices: an adventure in types. In *Proceedings of the International Conference on Functional Programming*, p. 28–35, 1999.

[11] S. Peyton Jones and J. Hughes (eds). *Haskell 98: A Non-Strict, Purely Functional Language.* `www.haskell.org/definition/`, 1999.