An initial-algebra approach to directed acyclic graphs

Jeremy Gibbons*

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand. Email: jeremy@cs.auckland.ac.nz

Abstract. The initial-algebra approach to modelling datatypes consists of giving *constructors* for building larger objects of that type from smaller ones, and *laws* identifying different ways of constructing the same object. The recursive decomposition of objects of the datatype leads directly to a recursive pattern of computation on those objects, which is very helpful for both functional and parallel programming.
We show how to model a particular kind of directed acyclic graph using this initial-algebra approach.
Keywords. Graphs, data types, catamorphisms, initial algebras, Bird-Meertens Formalism, program derivation.

1 Introduction

It is now widely recognized that the traditional ad-hoc approaches to program construction do not yield reliable software; a more systematic and formal approach is required. One such approach consists of *program verification*—proving after the fact that a given program satisfies its formal specification. This approach turns out to be difficult to implement, not least because the vast majority of programs would *not* satisfy their specification, even if they had one, but more importantly, because program verification gives no direct help in actually constructing the program in the first place.

An alternative approach is provided by *program derivation*, whereby a program is *calculated* from its formal specification by the application of a series of correctness-preserving transformations. The resulting program is guaranteed to satisfy its specification (assuming that the calculation is carried out correctly), but now its construction and verification are performed together, allowing insights from each to help with the other.

 This work was partially supported by University of Auckland Research Committee grant numbers A18/XXXXX/62090/3414013, /3414019 and /3414024.

Such a calculational approach necessitates having a body of *notations* for writing programs and *theorems* for proving equalities between them—that is, a *calculus* of programs. The Bird-Meertens Formalism [20, 5, 1] is one such calculus; it relies on tightly-coupled notions of *data* and *program* structure to yield its notations and theorems. In particular, datatypes are defined as extreme (initial or terminal) objects in categories of algebras—equivalently, extreme solutions of recursive systems of equations—and various morphisms representing common patterns of computation on those datatypes defined as the corresponding unique arrows from or to those objects. In this paper, we consider only initial algebras and *catamorphisms*, the corresponding morphisms; how well the ideas translate to final algebras and other morphisms remains to be seen.

Defining a datatype as an initial algebra essentially consists of giving two kinds of object:

- *constructors* for building larger elements of that type from smaller elements, and
- *laws* identifying syntactically different but semantically equivalent ways of constructing an element of that type.

Studying the initial algebra corresponding to a datatype gives new ways of implementing that datatype, and new insight into old algorithms—and sometimes even new algorithms—on that datatype. Moreover, the initial-algebra approach to datatypes appears to be particularly suitable for implementation in functional languages and in languages for parallel execution [29].

We have a good understanding of initial algebras corresponding to many common datatypes, such as lists [5], sets and bags [1, 14, 7], trees [21, 15, 12], and arrays [30, 6, 16]. One datatype ubiquitous in computing but conspicuous by its absence from this collection is that of *graphs*. The reason for this absence is that in order to model graphs, it appears that some means of 'naming' subcomponents is required. In contrast, the initial-algebra approach permits only 'structural' references to subcomponents.

In this paper we take steps towards remedying this absence, by defining and exploring an initial algebra corresponding to directed acyclic graphs. We show that naming is not necessary for modelling directed acyclic graphs. However, these are only the first steps; for one thing, the algebra does not correspond exactly to directed acyclic graphs, and for another, there are other kinds of graphs (for example, undirected graphs and directed cyclic graphs) to consider. These are topics for further study.

The rest of this paper is organized as follows. In Section 2, we review the initial-algebra approach to modelling datatypes. In Section 3, we present an initial-algebra definition of unlabelled directed acyclic graphs. In Section 4, we discuss catamorphisms on graphs. In Section 5, we generalize the construction of Section 3 to labelled directed acyclic graphs. In Section 6, we discuss other approaches to representing graphs in a style suitable for functional programming. Finally, Section 7 summarizes and presents directions for further work.

Throughout this paper, we write '.' for function application, which associates to the right, and '$\circ$' for function composition, which is associative:

$$(g \circ f).a = g.(f.a) = g.f.a$$

We write '$a : A$' for 'a has type A', and '$\mathbb{N}$' for the type of natural numbers including zero. For associative operator $\oplus$, we write $copy(n, \oplus, x)$ as an abbreviation for $x \oplus x \oplus \cdots \oplus x$ with n occurrences of x. For any x, $copy(0, \oplus, x)$ is the unit of $\oplus$ if it exists.

2 Initial Algebras and Catamorphisms

We introduce the initial-algebra approach to datatypes by way of a simple familiar example, the algebra of *join lists*. We use what may seem like unnecessarily heavy machinery for this simple example; the reason is that the machinery is necessary for the more complex algebra of directed acyclic graphs that is the subject of this paper.

2.1 Unlabelled Join Lists

We start by considering 'unlabelled join lists' (UJLs), which are finite possibly-empty chains of unlabelled nodes. For example, the UJL with three nodes might be drawn as in Figure 1. UJLs are built using three constructors: the constants *null* and *node*, respectively representing the empty list and the list with one node, and the binary operator $+\!\!+$ (pronounced 'join') which joins two lists to make a (usually) longer list.

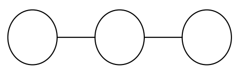

Fig. 1. An unlabelled join list

These constructors obey some laws, identifying different ways of building the same list. These laws are that $+\!\!+$ is associative and has unit *null*. (In other words, the constructors form a monoid.) For example, the list in Figure 1 is represented by (among others) the expression

$$node +\!\!+ node +\!\!+ node$$

Because of associativity, no parentheses are needed.

UJLs can be modelled as a category. Recall that a *category* consists of a collection of objects and a collection of arrows between objects. We write '$x : m \to n$' to indicate that arrow x goes from object m to object n. Compatible arrows can be composed; if $x : m \to n$ and $y : n \to p$ then $x;y : m \to p$.

Composition is associative, and for every object m there is an *identity arrow* $id_m : m \rightarrow m$ which is the unit of composition to or from that object.

In the case of UJLs, the category has a single object, corresponding to the type of all UJLs, and arrows corresponding to the lists themselves. Composition of arrows corresponds to joining lists; since there is but a single object, all pairs of arrows are composable. The identity arrow corresponds to the empty list. We require an arrow corresponding to the list *node* with a single element; since the collection of arrows is closed under composition, there is then necessarily an arrow corresponding to every UJL.

If we now consider those categories with a single object, they in turn form a category $\mathcal{L}$, with objects the categories in question and arrows the functors between these categories. (A *functor* F from category B to category C is a morphism on categories taking objects of B to objects of C and arrows of B to arrows of C such that, if $x : m \rightarrow n$ in B then $F.x : F.m \rightarrow F.n$ in C, and moreover $F.id_m = id_{F.m}$ and $F.(x;y) = F.x;F.y$.) We define the algebra of UJLs to be the *initial* object in $\mathcal{L}$, that is, the object in $\mathcal{L}$ from which there is a unique arrow to any other object in $\mathcal{L}$. (The initial object is unique up to isomorphism, and can be shown to exist.) Informally, this states that UJLs form the *smallest* algebra closed under the constructors in which the given laws hold, and no other laws do.

2.2 Labelled Join Lists

Of course, the type of UJLs is not very interesting; it is isomorphic to the natural numbers. We presented it simply because it happens to be the list-like algebra closest to the unlabelled directed acyclic graphs that we introduce later.

We can generalize UJLs to *labelled join lists* (LJLs) with nodes labelled by elements of some type A. The node constructor changes, so that now $node.a$ is a LJL for every $a : A$, but the two constructors *null* and $+\!\!+$ and the laws do not change. For example, the LJL in Figure 2 is represented by the expression

$$node.3 +\!\!+ node.1 +\!\!+ node.6$$

where the label type is $\mathbb{N}$.

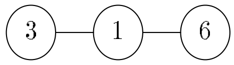

Fig. 2. A labelled join list

LJLs with labels of type A are by definition the initial object in the category consisting of categories with just one object and an arrow corresponding to every element of type A.

UJLs are isomorphic to LJLs with labels drawn from the unit type (the type with exactly one element), so from now on we will use the term 'join lists' to refer to LJLs.

2.3 Catamorphisms on Join Lists

Since the algebra of join lists was defined to be the initial object in the appropriate category, there is by definition a unique morphism from that algebra to any target algebra in the category. Such a morphism is called a *join list catamorphism*, and is uniquely determined by that target algebra.

Put another way, a function h from join lists with labels of type A to another type B is a join list catamorphism iff there exist a constant $b : B$, a function $f : A \to B$ and a binary operator $\oplus : B \times B \to B$ such that

$$\begin{aligned} h.null &= b \\ h.node.a &= f.a \\ h.(x +\!\!\!+\, y) &= h.x \oplus h.y \end{aligned}$$

and such that $\oplus$ is associative and has unit b. (The target category in this case has a single object B and arrows corresponding to elements of B; composition of arrows corresponds to $\oplus$, and the identity arrow is b.) We write '$(\![b, f, \oplus]\!)$' for such an h.

There are many examples of interesting join list catamorphisms. A few simple ones are as follows. The identity function on join lists is $(\![null, node, +\!\!\!+]\!)$. The function *length*, returning the number of nodes in a list, is $(\![0, f, +]\!)$ where $f.a = 1$ for each a. The function *reversel*, which reverses a join list, is $(\![null, node, \oplus]\!)$ where $t \oplus u = u +\!\!\!+\, t$.

3 Unlabelled Directed Acyclic Graphs

In this section, the main part of the paper, we present an initial-algebra definition of a particular kind of directed acyclic graph.

3.1 Directed Acyclic Multigraphs

The particular kind of graph we will model is that of directed acyclic graphs, but with a few unconventional aspects:

- there may be more than one edge between a given pair of vertices (thus, these are *multigraphs* rather than simply graphs)
- the incoming and outgoing edges of a vertex are ordered (that is, they form a sequence, rather than a bag or set)

– the graph as a whole has a sequence of incoming edges ('entries') with targets but no sources, and a sequence of outgoing edges ('exits') with sources but no targets; entries and exits are collectively called 'connections'

We call such a graph a *directed acyclic multigraph*, or DAMG (pronounced 'damage') for short.

For $m, n : \mathbb{N}$, the type $G_{m,n}$ consists of DAMGs with m entries and n exits. Thus, a graph of type $G_{3,4}$ has the form pictured in Figure 3. We write G for the type of all DAMGs.

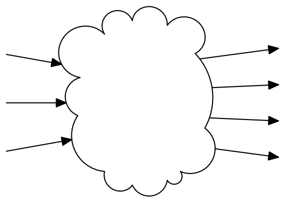

Fig. 3. The form of a graph of type $G_{3,4}$

3.2 Constructors

DAMGs are built from six constructors, as explained below.

Vertices. Vertices are represented by a set *vert* indexed by pairs of natural numbers, such that $vert_{m,n} : G_{m,n}$ for $m, n : \mathbb{N}$. The intention is that $vert_{m,n}$ represents a single vertex with m entries and n exits. For example, $vert_{3,2}$ might be drawn as in Figure 4.

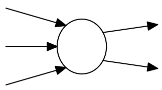

Fig. 4. The vertex $vert_{3,2}$

Edges. The constant $edge : G_{1,1}$ is simply an edge, with a single entry and a single exit. It would be drawn as in Figure 5.

Fig. 5. An edge

Beside. If $x : G_{m,n}$ and $y : G_{p,q}$ then $x \parallel y$ (pronounced 'x beside y') is of type $G_{m+p,n+q}$. Informally, $x \parallel y$ consists of x 'in parallel with' y; for example, $vert_{1,2} \parallel vert_{2,1}$ (of type $G_{3,3}$) might be drawn as in Figure 6. (In drawings of graphs, we order connections from top to bottom, and direct them from left to right.)

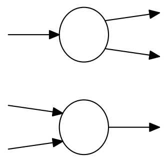

Fig. 6. $vert_{1,2} \parallel vert_{2,1}$

The constructor $\parallel$ is associative, so $x \parallel (y \parallel z) = (x \parallel y) \parallel z$. We write '$m \times x$' as an abbreviation for $copy(m, \parallel, x)$; we see later that $\parallel$ has a unit, so $0 \times x$ is defined.

Before. If $x : G_{m,n}$ and $y : G_{n,p}$, then $x \mathbin{;} y$ (pronounced 'x before y') has type $G_{m,p}$, and is formed by connecting the exits of x to the entries of y. For example, $vert_{0,1} \mathbin{;} vert_{1,0}$ might be drawn as in Figure 7.

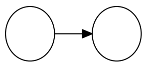

Fig. 7. $vert_{0,1} \mathbin{;} vert_{1,0}$

The constructor $;$ is associative; that is, $x \mathbin{;} (y \mathbin{;} z) = (x \mathbin{;} y) \mathbin{;} z$ if both expressions are correctly typed. (Note that if either expression is incorrectly typed, then both are.)

We write '$;_m$' for the restriction of $;$ to pairs of DAMGs with exactly m intermediate connections. Note that $;_m$ has unit $m \times edge$.

A further property enjoyed by $\parallel$ and $;$ is the so-called *abiding law*. If $w : G_{m,n}$, $x : G_{n,p}$, $y : G_{q,r}$ and $z : G_{r,s}$, then

$$(w \mathbin{;_n} x) \parallel (y \mathbin{;_r} z) = (w \parallel y) \mathbin{;_{n+r}} (x \parallel z)$$

For example,

$$(vert_{2,1} \mathbin{;} vert_{1,1}) \parallel (vert_{1,1} \mathbin{;} vert_{1,2}) = (vert_{2,1} \parallel vert_{1,1}) \mathbin{;} (vert_{1,1} \parallel vert_{1,2})$$

—in pictures, both sides might be drawn as in Figure 8. Notice that the type information is important here; without it, $(w \talloblong y) \fatsemi (x \talloblong z)$ may be well-typed when $(w \fatsemi x) \talloblong (y \fatsemi z)$ is not.

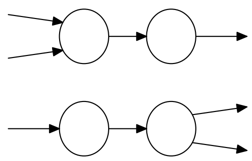

Fig. 8. An instance of the abiding law

The name 'abiding' is due to Bird [6]. He coined it as a contraction of 'above' and 'beside', operators which he used for building a larger array by putting one smaller array above or beside another.

Empty. We introduce a constructor *empty* for the empty graph, largely because of the elegant properties that it enjoys. It would be drawn as a blank picture. The empty graph satisfies the following two laws.

- *empty* is the unit of $\talloblong$ (and so, for any x, $0 \times x = \mathit{empty}$)
- *empty* (being $0 \times \mathit{edge}$) is also the unit of $\fatsemi_0$

From these we can conclude that, if $x : G_{m,0}$ and $y : G_{0,n}$, then $x \fatsemi_0 y = x \talloblong y$, since

$$
\begin{array}{ll}
 & x \fatsemi_0 y \\
= & \qquad \{\mathit{empty} \text{ is the unit of } \talloblong\} \\
 & (x \talloblong \mathit{empty}) \fatsemi_0 (\mathit{empty} \talloblong y) \\
= & \qquad \{\text{abiding}\} \\
 & (x \fatsemi_0 \mathit{empty}) \talloblong (\mathit{empty} \fatsemi_0 y) \\
= & \qquad \{\mathit{empty} \text{ is the unit of } \fatsemi_0\} \\
 & x \talloblong y
\end{array}
$$

For example, both $\mathit{vert}_{2,0} \fatsemi_0 \mathit{vert}_{0,1}$ and $\mathit{vert}_{2,0} \talloblong \mathit{vert}_{0,1}$ could be drawn as in Figure 9. We call this the *dislocation law.* Symmetrically, $x \fatsemi_0 y = y \talloblong x$.

Swap. The five constructors we have seen so far can construct only planar graphs. The constructor *swap* escapes from planarity. For $m, n \in \mathbb{N}$, $\mathit{swap}_{m,n}$ has type $G_{m+n,n+m}$, and consists of m edges connecting the first m entries to the last m exits, and n edges connecting the last n entries to the first n exits. For example, $\mathit{swap}_{3,2}$ has type $G_{5,5}$, and might be drawn as in Figure 10.

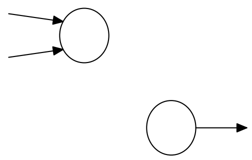

Fig. 9. An instance of the dislocation law

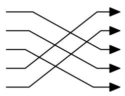

Fig. 10. $swap_{3,2}$

Swaps satisfy a number of laws. The first of these laws states that swapping zero connections makes no difference:

$$swap_{m,0} = m \times edge$$

The second law shows that swapping $n+p$ connections can be done by swapping n connections and then swapping p connections:

$$swap_{m,n+p} = (swap_{m,n} \parallel (p \times edge)) \fatsemi ((n \times edge) \parallel swap_{m,p})$$

The right-hand side of this equation is illustrated in Figure 11, in the case when $m = 1$, $n = 2$ and $p = 3$. We call these last two laws the *swap simplification laws*.

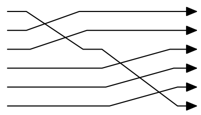

Fig. 11. Breaking down a larger *swap*

The third law relates swaps to other constructs. If $x : G_{n,p}$ and $y : G_{m,q}$ then

$$swap_{m,n} \fatsemi (x \parallel y) \fatsemi swap_{p,q} = y \parallel x$$

We call this the *swap law*. The left-hand side of this equation is illustrated in Figure 12, in the case when $x = vert_{2,1}$ and $y = vert_{1,2}$; then the right-hand side is as in Figure 6.

In the special case when $n = p$, $x = n \times edge$, $m = q$ and $y = m \times edge$, the swap law simplifies to

$$swap_{m,n} \fatsemi swap_{n,m} = (m + n) \times edge$$

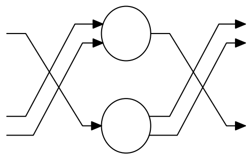

Fig. 12. An instance of the swap law

From this law and the earlier simplification laws we can deduce simplification laws for the first index too:

$$swap_{0,n} = n \times edge$$

and

$$swap_{m+n,p} = ((m \times edge) \mathbin{[\!]} swap_{n,p}) \mathbin{⨾} (swap_{m,p} \mathbin{[\!]} (n \times edge))$$

Note that, in view of the swap simplification laws, any swap can be built from $swap_{1,1}$ and edges using $[\!]$ and $⨾$, so in that sense we could replace the family of swap constructors with just $swap_{1,1}$. However, it appears that the general form of the swap law is then difficult to express.

3.3 An Example Graph

As an example, we show how to construct the graph in Figure 13.

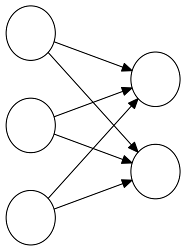

Fig. 13. An example graph

'Teasing' apart the edges, we see that this graph is equivalent to the exploded graph in Figure 14. Hence the graph is represented by the expression

$$(3 \times vert_{0,2}) \mathbin{⨾} (edge \mathbin{[\!]} ((2 \times swap_{1,1}) \mathbin{⨾} (edge \mathbin{[\!]} swap_{1,1} \mathbin{[\!]} edge)) \mathbin{[\!]} edge) \mathbin{⨾} (2 \times vert_{3,0})$$

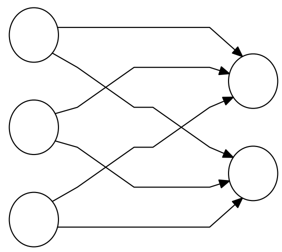

Fig. 14. An 'exploded view' of the graph in Figure 13

3.4 DAMGs As Symmetric Strict Monoidal Categories

It turns out that the algebra of DAMGs is essentially a *symmetric strict monoidal category* [19] enriched with objects representing vertices. We expand here on this observation.

A *strict monoidal category* (SMC) $(B, +, e)$ is a category B in which:

- the objects of B form a monoid with respect to $+$ (as a binary operation on objects of B) and e (as an object of B)
- the operator $+$ also acts on arrows of B; if $x : m \to n$ and $y : p \to q$ then $x + y : m + p \to n + q$; moreover, $+$ satisfies the laws

$$\begin{aligned} (x+y)+z &= x+(y+z) \\ id_e + x &= x \\ x + id_e &= x \\ id_m + id_n &= id_{m+n} \\ (w+x);(y+z) &= (w;y)+(x;z) \end{aligned}$$

provided in the last case that all the compositions are defined.

A *symmetric strict monoidal category* (SSMC) $(B, +, e, \gamma)$ is a SMC $(B, +, e)$ with a family of arrows $\gamma_{m,n} : m + n \to n + m$ for all objects m and n of B, for which the following laws hold:

$$\begin{aligned} \gamma_{m,0} &= id_m \\ \gamma_{m,n+p} &= (\gamma_{m,n} + id_p);(id_n + \gamma_{m,p}) \\ \gamma_{m,n};(x+y);\gamma_{p,q} &= y + x \end{aligned}$$

provided in the last case that $x : n \to p$ and $y : m \to q$.

Clearly, the algebra of DAMGs forms a SSMC $(B, [\![, empty, swap)$ in which the category B has as objects the natural numbers, and arrows $x : m \to n$ corresponding to DAMGs x in $G_{m,n}$. Composition of arrows is $\fatsemi$, and the identity object on m is $m \times edge$.

Now consider SSMCs in which the objects of the base category B are the natural numbers, and the collection of arrows of B also contains arrows $v_{m,n} : m \to n$ for each pair of naturals m, n. We call such a SSMC an *enriched* SSMC, and write $\mathcal{G}$ for the category of all enriched SSMCs (with functors between SSMCs as arrows). We define the algebra of DAMGs to be the *initial* SSMC in the category $\mathcal{G}$. Informally, this says that DAMGs form the smallest algebra closed under the constructors in which all and only the DAMG laws hold.

3.5 Soundness and Completeness of the Laws

When axiomatizing a datatype, it is usually obvious whether sufficient constructors have been chosen to represent all elements of the intended model. If there are not enough constructors, extra ones can be added as necessary, and the worst that can happen is some redundancy in the resulting datatype.

It is more difficult to tell whether the right collection of laws has been chosen, since this collection must be neither too strong nor too weak. The collection must satisfy the following two properties.

Soundness: The given collection of laws must certainly be true of the intended model. That is, the laws must not be too strong, identifying distinct elements of the intended model.

Completeness: Soundness can be attained simply by having no laws at all. The competing requirement is that the collection of laws must be complete. That is, the laws must be sufficient to identify any two representations of the same element in the intended model. In other words, the collection of laws must also not be too weak.

We have just seen that the five constructors *edge*, $⨾$, $[\!]$, *empty* and *swap*, together with all the laws (that is, the whole algebra except the vertices), form exactly a SSMC in which the objects are the natural numbers. Căzănescu and Ştefănescu [10] show that such a category axiomatizes bijective relations; bijective relations are the initial algebra with those five constructors and those laws.

Since none of the laws involve the vertices, the whole algebra (all six constructors together with the laws) axiomatizes vertices with bijections for 'plumbing' between them. This is clearly exactly the datatype of directed acyclic multigraphs; the laws we have defined are indeed sound and complete.

4 DAMG Catamorphisms

We defined the algebra of DAMGs to be the initial object in the category $\mathcal{G}$ of enriched SSMCs. By definition, therefore, there is a unique morphism from the algebra of DAMGs to any other enriched SSMC. Such a morphism is called a DAMG *catamorphism*, and is uniquely determined by that other enriched SSMC.

Put another way, a function $h : G \to B$ is a DAMG catamorphism iff there exist constants $a, b : B$, families of constants $v_{m,n} : B$ and $s_{m,n} : B$ indexed by pairs of naturals m, n, and binary operators $\oplus : B \times B \to B$ and $\otimes : B \times B \to B$

such that

$$\begin{aligned} h.\mathit{empty} &= a \\ h.\mathit{edge} &= b \\ h.\mathit{vert}_{m,n} &= v_{m,n} \\ h.\mathit{swap}_{m,n} &= s_{m,n} \\ h.(x \mathbin{[\!]} y) &= h.x \oplus h.y \\ h.(x \mathbin{;} y) &= h.x \otimes h.y \end{aligned}$$

(in fact, $h.x \otimes h.y$ need only be defined when x and y are compatible) and such that these constants and functions form an enriched SSMC in the obvious way. We write '$(\![a, b, v, s, \oplus, \otimes]\!)$' for such an h; these six items uniquely determine h.

4.1 Examples of DAMG Catamorphisms

Some simple examples of DAMG catamorphisms are as follows. The identity function on G is $(\![\mathit{empty}, \mathit{edge}, \mathit{vert}, \mathit{swap}, [\!], ;]\!)$. The function *nvertices*, which returns the number of vertices, is $(\![0, 0, 1, 0, +, +]\!)$. (We write simply '1' for the family of constants indexed by pairs of naturals, each member of which is 1.) The function *reverseg*, which reverses a DAMG, is $(\![\mathit{empty}, \mathit{edge}, \mathit{vert}, s, [\!], \otimes]\!)$ where $s_{m,n} = \mathit{swap}_{n,m}$ and $t \otimes u = u \mathbin{;} t$.

A more interesting example is the function *sp*, which returns the length of the shortest path from each entry to each exit; *sp* takes a DAMG of type $G_{m,n}$ and returns an $m \times n$ matrix of values in $\mathbb{N} \cup \{\infty\}$. We have

$$sp = (\![a, b, v, s, \oplus, \otimes]\!)$$

where

- a is the 0×0 matrix
- b is the 1×1 matrix containing a 0
- $v_{m,n}$ is the $m \times n$ matrix consisting entirely of 1s
- $s_{m,n}$ is the $(m+n) \times (n+m)$ matrix of the form $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ in which A and D are $m \times n$ and $n \times m$ submatrices consisting entirely of ∞s, and B and C are $m \times m$ and $n \times n$ submatrices with zeroes on the leading diagonals and ∞s elsewhere
- if t and u are $m \times n$ and $p \times q$ matrices, respectively, then $t \oplus u$ is the $(m+p) \times (n+q)$ matrix $\begin{pmatrix} t & \infty \\ \infty & u \end{pmatrix}$—that is, with elements from t and u in the top left and bottom right quadrants, and ∞ filling the other two quadrants
- if t and u are $m \times n$ and $n \times p$ matrices, respectively, then $t \otimes u$ is the matrix product of t and u in the closed semiring $(\mathit{min}, +)$—that is,

  $$(t \otimes u)_{i,j} = \mathop{\mathit{min}}_{1 \leq k \leq n} (t_{i,k} + u_{k,j}) \qquad \text{for } 1 \leq i \leq m,\ 1 \leq j \leq p$$

  where ∞ is the zero of addition and the unit of *min*.

For example, the DAMG in Figure 15 is represented by the expression:

$$vert_{1,2} \fatsemi ((vert_{1,1} \fatsemi vert_{1,1}) \mathbin{[\!]} vert_{1,1}) \fatsemi vert_{2,1}$$

and one way of computing its single shortest path could be as illustrated in Figure 16. Thus, the shortest path between the connections of $vert_{1,1}$ has just one vertex, and that between the connections of $vert_{1,1} \fatsemi vert_{1,1}$ has two; the shortest paths between the four possible pairs of connections of $(vert_{1,1} \fatsemi vert_{1,1}) \mathbin{[\!]} vert_{1,1}$ have lengths 2, ∞, ∞ and 1. The shortest path from the only entry to the only exit of the whole graph has three vertices.

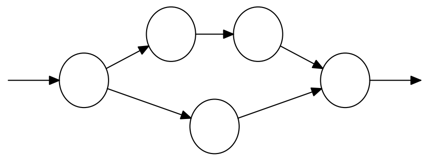

Fig. 15. Another example graph

$$\underbrace{\underbrace{\underbrace{vert_{1,2}}_{(1\ 1)} \fatsemi (\underbrace{(\underbrace{\underbrace{vert_{1,1}}_{(1)} \fatsemi \underbrace{vert_{1,1}}_{(1)}}_{(2)}) \mathbin{[\!]} \underbrace{vert_{1,1}}_{(1)}}_{\begin{pmatrix}2 & \infty\\ \infty & 1\end{pmatrix}})}_{(3\ 2)} \fatsemi \underbrace{vert_{2,1}}_{\begin{pmatrix}1\\ 1\end{pmatrix}}}_{(3)}$$

Fig. 16. The shortest path between connections of the graph in Figure 15

We should check that sp really is a DAMG catamorphism, that is, that the six components really do form an enriched SSMC. We leave it to the reader to verify (writing i_m for $copy(m, \oplus, b)$, the $m \times m$ matrix with 0s on the leading diagonal and ∞s elsewhere) that:

- $\oplus$ is associative, and has unit a
- $\otimes$ is associative, and has unit i_m (for suitable value of m)
- $(w \otimes x) \oplus (y \otimes z) = (w \oplus y) \otimes (x \oplus z)$ for compatible matrices w, x and y, z
- $s_{m,0} = i_m$
- $s_{m,n+p} = (s_{m,n} \oplus i_p) \otimes (i_n \oplus s_{m,p})$

– $s_{m,n} \otimes (x \oplus y) \otimes s_{p,q} = y \oplus x$ for $n \times p$ matrix x and $m \times q$ matrix y

If still keen after doing so, the reader may also wish to verify that the function that computes the *longest* path between any pair of connections is also a catamorphism.

5 Labelled DAMGs

In this section we discuss labelling the vertices and edges of a DAMG.

5.1 Vertex-Labelled DAMGs

We can generalize to vertex-labelled DAMGs easily. We write $G_{m,n}.A$ for the type of DAMGs with m entries and n exits and vertices labelled with elements of A, and $G.A$ for the type of vertex-labelled DAMGs with any number of connections. Then, for a of type A, $vert_{m,n}.a$ is of type $G_{m,n}.A$, and consists of a single vertex with m entries and n exits and label a. The other five constructors and all the laws remain unchanged.

Thus, the expression

$$vert_{1,2}.3 \fatsemi ((vert_{1,1}.2 \fatsemi vert_{1,1}.5) \parallel vert_{1,1}.9) \fatsemi vert_{2,1}.7$$

represents the vertex-labelled DAMG of type $G_{1,1}.\mathbb{N}$ in Figure 17.

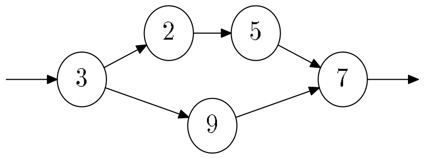

Fig. 17. A vertex-labelled DAMG

5.2 Edge-Labelled DAMGs

It is more difficult to model edge-labelled DAMGs satisfactorily. For example, should we label all connections? If so, what happens when connections matched by $\fatsemi$ do not have the same label? Should $\fatsemi$ be a partial operator, undefined in such cases? Or should it be asymmetric, taking (say) the labels from its first argument? Alternatively, we could label only 'complete edges'—edges with a vertex at each end—and leave 'dangling' connections unlabelled; then $\fatsemi$ could take also a list of of the appropriate number of labels with which to label connections. Another alternative would be to label the connections with elements of a monoid (for example, lists), and combine the labels on matched connections using the binary operation of the monoid.

It is not at all clear which is the best approach to take.

5.3 Topological Sort

One operation suitable for vertex-labelled DAMGs is topological sort; given a DAMG, return the vertex labels as a list whose ordering respects the edge ordering of the graph. Is topological sort a catamorphism?

It would appear so. Topological sort ts satisfies the following properties.

- $ts.empty$, $ts.edge$ and $ts.swap_{m,n}$ are all just $null$, since these graphs have no vertices
- $ts.vert_{m,n}.a$ is $node.a$
- $ts.(x \mathbin{;} y)$ is $ts.x +\!\!+ ts.y$
- $ts.(x \parallel y)$ is any interleaving of $ts.x$ and $ts.y$—for example, $ts.x +\!\!+ ts.y$

In other words, we can topologically sort a DAMG by deleting from the expression by which it was constructed everything except the labels; this necessarily gives the correct labels in a correct order.

Unfortunately, things are not so straightforward. In general, a DAMG has many topological sorts, but the function ts can return only one of them. Moreover, with the way we have defined ts above, the particular topological sort returned will depend on the way that the graph was constructed. For example, suppose that we take $ts.(x \parallel y) = ts.x +\!\!+ ts.y$, as suggested above. Then the two graphs

$$(vert_{1,1}.1 \mathbin{;} vert_{1,1}.2) \parallel (vert_{1,1}.3 \mathbin{;} vert_{1,1}.4)$$

and

$$(vert_{1,1}.1 \parallel vert_{1,1}.3) \mathbin{;} (vert_{1,1}.2 \parallel vert_{1,1}.4)$$

(which by the abiding law are equal) will have different images, $[1, 2, 3, 4]$ and $[1, 3, 2, 4]$, under ts. Both images are valid topological sorts of the graph, but if ts is to be well-defined it must return exactly the same topological sort as result given the same graph as argument.

Put another way, the sextuple of components $(null, null, node, null, +\!\!+, +\!\!+)$ does not form an enriched SSMC, since it does not satisfy all the DAMG laws. Neither does $(null, null, node, null, \oplus, +\!\!+)$ for any $\oplus$ such that $x \oplus y$ is an interleaving of x and y; in particular, $+\!\!+$ does not abide with any deterministic interleave operator. The problem is that a single topological sort of each of x and y is sufficient information to compute *one* topological sort of $x \mathbin{;} y$, but not in general to compute *all* topological sorts. Topological sort is not a DAMG catamorphism.

(The problem appears to do with the deterministic interleaving for $ts.(x \parallel y)$, which suggests that although topological sort is not a *functional* catamorphism, it might be a *relational* catamorphism [2]. The topological sorts of $x \parallel y$ would be *any* interleaving of the topological sorts of x and of y. Unfortunately, given topological sorts s and t of x and y, still the only list guaranteed to be a topological sort of $x \mathbin{;} y$ is $s +\!\!+ t$—although other interleavings of s and t may also be. The two different representations of the same graph above will still have different topological sort relations—the first representation allows $[1, 2, 3, 4]$ whereas the second does not. Intuitively, the non-determinism is 'too local'; it turns out that 'more global' non-determinism is needed. In fact, the function that returns the set of *all* topological sorts of a DAMG *is* a functional catamorphism [27].)

6 Other Approaches to Modelling Graphs

In this section we discuss a number of other approaches to modelling graphs, and compare them to the initial-algebra approach presented here.

6.1 Traditional Representations

Graphs are traditionally represented in one of three ways:

- a set of vertices and a set of edges
- a collection of adjacency lists
- an adjacency matrix

None of these representations are particularly suitable for implementing graph algorithms in a functional language. More to the point, however, none of these representations recursively composes larger graphs out of smaller ones, and so none of them provides for free a pattern of computation on graphs that recursively decomposes its argument into smaller graphs. Such patterns of computation—catamorphisms—seem very useful for functional and parallel programming.

6.2 Graphs in Functional Languages

Directed graphs can be represented in a lazy functional language using cyclic data structures [4]. For example, the Gofer definition

```
[node1,node2,node3] where node1 = (1, [node2,node3])
                          node2 = (2, [node1])
                          node3 = (3, [])
```

represents the cyclic graph in Figure 18 as a list of vertices where each vertex is a pair consisting of a label and an adjacency list. The disadvantage of this approach is that a cyclic graph is operationally indistinguishable from an infinite tree. Kashiwagi and Wise [17] use this approach to implement some graph algorithms (strong components, connected components, acyclicity) by having a stream of 'updateable' values at each node and a problem-specific method of finding fixed points on those streams. This produces a graph labelled with results, which, if cyclic, is again indistinguishable from an infinite tree.

A related approach [28] is to represent the graph as a function of type $N \rightarrow F.N$, where N is the type of node identifiers and F is some functor.

King and Launchbury [18] implement some graph algorithms (topological sort, connected components, strong components, reachability) by imperatively performing a depth-first search on the graph, and declaratively manipulating the resulting depth-first-search forest.

Burton and Yang [8] implement graphs in a pure functional language effectively by implementing an imperative store and threading this through the program.

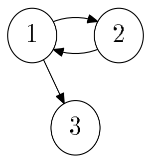

Fig. 18. A cyclic graph

6.3 Formal Languages and Relations

Möller [24, 25, 26] uses formal languages, and in particular multiary relations, to model graphs. He derives a number of graph algorithms, such as reachability, topological sort and cycle detection.

This approach gives concise specifications and calculations. However, graphs are still modelled monolithically—there is no recursive decomposition, and so no direct help in that way in constructing programs. (Help does come from another direction, though: from familiar properties of relations and formal languages.)

6.4 Graph Grammars

There is a large body of work in the field of *graph grammars*. Courcelle [11] gives definitions in terms of directed *hypergraphs*, in which edges may have arbitrarily many endpoints; to avoid too much extra notation, we discuss here just the specialization to edges with exactly two endpoints.

A graph has a *source*, a sequence consisting of some of the vertices of the graph (perhaps with omissions and duplication). Vertices in the source are 'external' and are available for connection to other graphs; other vertices are 'internal' and are hidden.

There are five constructors:

vertex: a single vertex, which is the sole element of the source
edge: a single edge, with two vertices that form the source
disjoint union: combines two graphs into a larger graph, concatenating the sources
source fusion: takes a graph and an equivalence relation δ on its sources, and identifies the vertices equivalent under δ
source redefinition: rearranges the source of a graph (perhaps omitting some vertices and duplicating others) according to a given mapping

There are eleven laws; these are sufficient to transform any term built from the above constructors into a (non-unique) normal form consisting of the disjoint union of some vertices and edges, submitted to a single source fusion and then a single source redefinition.

Graph grammars are appropriate as a basis for describing graph rewriting systems, but they seem less so for more general graph algorithms.

6.5 Skillicorn's Definition

Skillicorn [29] defines an algebra of connected undirected vertex-labelled graphs, using three constructors:

- an injection, mapping labels to vertices,
- a binary operator 'connect', connecting two disjoint graphs with a single edge, and
- a unary operator 'close', adding an edge to a graph, thereby creating a cycle.

In order to indicate which two vertices are connected by the 'connect' or 'close' operators, Skillicorn says that the two constructors 'are drawn as simple straight lines connecting the two vertices', which seems to imply that his graphs can be represented faithfully only by two-dimensional pictures, and not by one-dimensional terms in an algebra.

Moreover, Skillicorn does not state the laws needed to distinguish this algebra from an algebra of trees in which each node can have zero, one or two children. He is therefore forced to decompose a graph in exactly the same way as it was built, precluding any attempt at load-balancing for parallel execution.

6.6 Free Net Algebras

Molitor [22] defines an algebra of 'nets', modelling VLSI circuits. The constructors of this algebra are:

- a collection of basic 'cells',
- some wiring components (straight wires, corners, T-junctions and a 'crossover'), and
- two partial binary operators 'above' and 'beside' which compose circuit diagrams vertically and horizontally, provided that the edges to be matched have the same number of connections.

He gives a collection of fourteen rather complex laws, and claims that they are sound and complete. (The proof is omitted from Molitor's paper, and the reader referred to his thesis.)

This work may lead to an algebra of undirected hypergraphs, in which an 'edge' connects arbitrarily many vertices.

7 Conclusions

7.1 Summary

We have presented an initial algebra modelling a particular (and rather unconventional) kind of directed acyclic graph. We have shown that quite a few natural functions on these graphs are catamorphisms on the algebra we have defined; we have also seen one natural function (topological sort) that appears not to be a catamorphism. (We believe that this is no fault of the particular algebra presented here, but is inherent in any initial-algebra model of directed acyclic graphs.) We have also discussed a number of other approaches to representing graphs.

7.2 Further Work

There are several directions for further work that appear quite promising. A few of these are outlined below.

- One question that remains to be answered is whether the algebra presented here is practically useful. Many natural simple problems on DAMGs turn out to be DAMG catamorphisms, but we have not yet seen any more complicated problems whose solution was simplified by this algebra of DAMGs.
- A problem with the algebra we have defined here is that it does not model directed acyclic graphs particularly closely. We have had to introduce 'connections' (incoming edges with targets but no sources, and outgoing edges with sources but no targets) for the whole graph, allow multiple edges between a pair of vertices, and consider the ordering of the incoming and outgoing edges of a vertex to be significant, all in order to come up with an algebra at all. Is it possible to adapt this approach to yield an algebra that more closely models directed acyclic graphs?
- The 'symmetric strict monoidal category' approach we have used here is based heavily on the work of Căzănescu and Ştefănescu. They use it to obtain initial-algebra models of sixteen classes of finite relations, corresponding to all sixteen combinations of totality, surjectivity, univocality (that is, being single-valued or functional) and injectivity [10]. They go on [9] to present an algebraic theory of 'flownomials'—flowcharts abstracted on both the individual statements and the interconnection pattern; the algebra of cyclic flownomials consists of the algebra of acyclic flownomials (similar to our DAMGs) endowed with 'feedback' operator that cyclically connects the first few exits to the corresponding number of entries. This may present a way to adapt our approach to model also possibly cyclic graphs.
- Modelling undirected graphs appears to be more difficult, because vertex connections are not partitioned into two groups according to direction, and it is therefore less obvious how to connect subgraphs together.
- With all the other initial-algebra definitions of datatypes that have been explored to date, the concept of an *accumulation* has proved to be very powerful [13]. Essentially, an accumulation records all the partial results from the computation of a catamorphism. One application of 'forwards and backwards accumulations' on directed acyclic graphs might be to compute 'earliest and latest possible finishing times' for tasks in a project, in which the tasks are represented by the vertices of a graph (labelled with task duration) and their dependencies by the edges. However, all these other initial algebras have been 'free', that is, with no laws. It is not immediately obvious how to define accumulations on types with laws, since for these there may be different ways of representing the same object, and hence different ways of computing the same catamorphism on that object. These different computations necessarily return the same results, but may well do so with different collections of partial results; which computation should the accumulation record?

7.3 Acknowledgements

The author wishes to thank Bob Paige for pointing out the dislocation law, and other members of IFIP WG2.1 and the anonymous referees for many helpful comments. Also, this presentation would have been a lot less elegant without the help of Virgil Căzănescu and Gheorghe Ştefănescu's work on SSMCs.

References

1. Roland Backhouse. An exploration of the Bird-Meertens formalism. In *International Summer School on Constructive Algorithmics, Hollum, Ameland.* STOP project, 1989. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.
2. Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude. Relational catamorphisms. In Möller [23], pages 287–318.
3. R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors. *LNCS 669: Mathematics of Program Construction.* Springer-Verlag, 1993.
4. Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.
5. Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
6. Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science.* Springer-Verlag, 1988. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
7. Alex Bunkenburg. The Boom hierarchy. In Kevin Hammond and John T. O'Donnell, editors, *1993 Glasgow Workshop on Functional Programming.* Springer, 1993.
8. F. Warren Burton and Hsi-Kai Yang. Manipulating multilinked data structures in a pure functional language. *Software—Practice and Experience*, 20(11):1167–1185, November 1990.
9. Virgil Emil Căzănescu and Gheorghe Ştefănescu. Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, XIII:171–210, 1990.
10. Virgil-Emil Căzănescu and Gheorghe Ştefănescu. Classes of finite relations as initial abstract data types I. *Discrete Mathematics*, 90:233–265, 1991.
11. Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5. Elsevier, 1990.
12. Jeremy Gibbons. *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.
13. Jeremy Gibbons. Upwards and downwards accumulations on trees. In Bird et al. [3], pages 122–138. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.
14. Paul Hoogendijk. Relational programming laws in the Boom hierarchy of types. In Bird et al. [3], pages 163–190.

15. Johan Jeuring. Deriving algorithms on binary labelled trees. CWI, Amsterdam, July 1989.
16. Johan Jeuring. The derivation of hierarchies of algorithms on matrices. In Möller [23], pages 9–32.
17. Yugo Kashiwagi and David S. Wise. Graph algorithms in a lazy functional programming language. Technical Report 330, Department of Computer Science, Indiana University, April 1991.
18. David J. King and John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. Department of Computer Science, University of Glasgow, 1993.
19. Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.
20. Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.
21. Lambert Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.
22. Paul Molitor. Free net algebras in VLSI-theory. *Fundamenta Informaticae*, XI:117–142, 1988.
23. B. Möller, editor. *IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications.* North-Holland, 1991.
24. Bernard Möller. Derivation of graph and pointer algorithms. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, pages 123–160. Springer-Verlag, 1993.
25. Bernhard Möller. Algebraic calculation of graph and sorting algorithms. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *LNCS 735: Formal Methods in Programming and Their Applications*, pages 394–413. Springer-Verlag, 1993.
26. Bernhard Möller and Martin Russling. Shorter paths to graph algorithms. In Bird et al. [3], pages 250–268.
27. Bob Paige. Comment at IFIP Working Group 2.1 meeting, Renkum, January 1994.
28. Ross Paterson. Interpretations of term graphs. Draft. Department of Computing, Imperial College, 1994.
29. David B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.
30. Chris J. Wright. A theory of arrays for program derivation. Transferral dissertation, Programming Research Group, Oxford University, 1988.

This article was processed using the LaTeX macro package with LLNCS style