Higher Dimensional Trees, Algebraically

Neil Ghani[1]* Alexander Kurz[2]**

[1] University of Nottingham
[2] University of Leicester

Abstract. In formal language theory, James Rogers published a series of innovative papers generalising strings and trees to higher dimensions.Motivated by applications in linguistics, his goal was to smoothly extend the core theory of the formal languages of strings and trees to these higher dimensions.

Rogers' definitions focussed on a specific representation of higher dimensional trees. This paper presents an alternative approach which focusses more on their universal properties and is based upon category theory, algebras, coalgebras and containers. Our approach reveals that Rogers' trees are canonical constructions which are also particularly beautiful. We also provide new theoretical results concerning higher dimensional trees. Finally, we provide evidence for our devout conviction that clean mathematical theories provide the basis for clean implementations by showing how our abstract presentation makes computing with higher dimensional trees easier.

1 Introduction

Strings occur in the study of formal languages where they are used to define complexity classes such as those of regular expressions, context free languages, context sensitive languages etc. Trees also play a multitude of different roles and are often thought of as 2-dimensional strings. For instance, there is a clear and well defined theory of tree automata, of tree transducers and other analogues of string-theoretic notions [6]. Indeed, the recent interest in XML and its focus on 2-dimensional data has brought the formal language theory of trees to a wider audience.

In a series of innovative papers (see [11] and references therein), James Rogers asked how one can formalise, and hence extend, the idea that trees are two-dimensional strings to higher dimensions. The desire to *go up a dimension* is very natural - for example a parser will turn a string into a tree. Thus higher dimensional trees will certainly arise when parsing 2-dimensional trees and, more generally, when trees are considered not as part of the meta-theory of the formal languages of strings, but as objects worthy of their own study. Rogers came from a background in both formal languages and natural languages and his motivation to study higher dimensional trees was rooted in the use of the latter to study the former. For example, his paper discusses applications to Tree Adjoining Grammars, Government Binding Theory, and Generalised Phrase Structure Grammars.

Rogers' work was highly imaginative and he certainly had great success in generalising formal language theory from strings and trees to higher dimensions. However, his approach to

*

** Partially supported by EPSRC EP/C014014/1.

higher dimensional trees is very concrete and this makes his work notationally more cumbersome than one might prefer. For example, Rogers defines a tree as a *tree domain*, ie a set of paths satisfying the left-sibling and ancestor properties. Similarly, he defines higher dimensional trees to be sets of higher dimensional paths satisfying higher dimensional versions of the ancestor and left-sibling properties. These conditions are notationally quite cumbersome at the two dimensional level and this complexity is magnified at higher dimensions. This has practical consequences as it is our belief that clean mathematical foundations are required for clean implementations of both higher dimensional trees as data structures and the algorithms which manipulate them. In particular, implementing higher dimensional trees as higher dimensional tree domains involves the (potential) requirement to regularly verify that algorithms preserve the well-formedness condition of the set of higher dimensional paths in a higher dimensional tree domain.

We provide a more abstract treatment of higher dimensional trees where the fundamental concept is not the path structure of tree domains but rather the notion of fixed point and initial algebra. When viewed through this categorical prism, Rogers' definitions and constructions become very succinct and elegant. This is a tribute to both the sophistication of category theory in capturing high level structure and also to Rogers' insight in recognising these structures as being of fundamental mathematical and computational interest. The overall contributions of this paper are thus as follows:

- We provide a categorical reformulation of the definition of Rogers higher dimensional trees. Remarkably, the central construction in our reformulation is the hitherto unused quadrant of the space whose other members are the free monad, the completely iterative monad, and the cofree comonad.
- To demonstrate that this research has both practical as well as theoretical insight, we use this reformulation to show that classical results of Arbib and Manes on 'Machines in a category' apply to higher-dimensional automata. In particular, this gives procedures of determinisation and minimisation.
- In a similar vein, we show that while clearly being comonadic, higher dimensional trees are also monadic in nature. This is an example of the kind of result that is both fundamental and would be missed without the abstract categorical formulation.
- We justify our belief that clean mathematical foundations leads to a clean computation structure by implementing higher dimensional trees in the Haskell programming language.

Our intention with this research is to try to synthesise our abstract approach with the intuitions and applications of Rogers. This paper is just the beginning and we welcome feedback from our own community before involving people who work higher dimensional trees and natural languages. Connecting category theory, especially algebras and coalgebras, with other scientific disciplines is an important and valuable goal if our ideas are to spread and we are also to be open to influence from those outside of our field. To summarise what this paper offers, we believe that our use of category theory tames the apparent complexity which Rogers' definitions possess at first sight.

The paper is structured as follows. *Section 2* follows parts of Rogers [11] and presents his notions of higher-dimensional trees and automata. *Section 3* presents our reformulation of Rogers' notions using fixed-point equations and coalgebras. *Section 4* shows that Rogers' higher dimensional trees are examples of *containers* which allows us to deduce several useful meta-theoretic results needed later. *Section 5* defines a notion of deterministic higher dimensional automaton and shows that the classical theorem of determinisation and minimisation from automata theory hold.

Acknowledgements The 2nd author wishes to thank Ichiro Hasuo for helpful discussions.We are also grateful to the referees for their numerous comments that helped us to improve the presentation.

2 Rogers's Higher Dimensional Trees

The most pervasive definition of (finitely branching) trees is via the notion of a tree domain. A tree domain is an enumeration of the paths in a tree - since a path is a list of natural numbers, a tree domain is a subset of lists of natural numbers. However, there should be two conditions on sets of paths reflecting the fact that i) if a node has an $n{+}1$'th child, then there should be an n'th child; and ii) all nodes apart from the root have a parent. Thus tree domains are defined as follows

Definition 2.1 (Tree Domains). *A tree domain $T \subseteq \mathbb{N}^*$ is a subset of lists of natural numbers such that*

- *(LS): If $w.(n{+}1) \in T$, then $w.n \in T$*
- *(A): If $w.n \in T$, then $w \in T$*

We use . for the concatenation of a list with an element. We call the first condition the left-sibling property (LS) and we call the second condition the ancestor property (A). Notice how tree domains, by focusing on paths, will inevitably lead to a process of computation dominated by the creation and consumption of sets of paths satisfying (LS) and (A). As we shall see later, tree domains and the paths in them can be treated more abstractly, and in a cleaner fashion, by the shapes and positions of the container reformulation of tree domains.

However, for now, we want to ask ourselves how the tree domains given above can be generalised from being 2-dimensional structures to n-dimensional structures. In the 2-dimensional case we had a notion of path as a list of natural numbers and then a tree domain consisted of a set of paths satisfying the properties (LS) and (A). Rogers defines n-dimensional tree domains by first defining what an n-dimensional path is and then defining an n-dimensional tree domain to be a set of n-dimensional paths satisfying higher dimensional variants of (LS) and (A). So what is an n-dimensional path? Notice that a natural number is a list of 1s and hence a list of natural numbers is a list of lists of 1s. Thus

Definition 2.2 (Higher Dimensional Paths [11, Def 2.1]). *The n-dimensional paths form a $\mathbb{N}$-indexed set P with $P_0 = 1$ (the one element set) and with P_{n+1} defined to be the least set satisfying*

- $[] \in P_{n+1}$
- *If $[x_1, .., x_m] \in P_{n+1}$ and $x \in P_n$, then $[x_1, .., x_m, x] \in P_{n+1}$*

A simpler definition would be that $P_n = \mathsf{List}^n 1$ but we wanted to give Roger's definition to highlight its concreteness. Having defined the n-dimensional paths we can define the n-dimensional tree domains as follows

Definition 2.3 (Higher Dimensional Tree domains [11, Def 2.2]). *Let $T_0 = \{\emptyset, 1\}$. The set T_{n+1} of $n+1$-dimensional tree domains consists of those subsets $T \subseteq P_{n+1}$ such that*

– *(HDLS): If $s \in P_{n+1}$, then $\{w \in P_n | s.w \in T\} \in T_n$*
– *(HDA): If $s.w \in T$, then $s \in T$*

The first condition is the higher dimensional left sibling property (HDLS). It is slightly tricky as, in higher dimensions, there is no unique left sibling and so one cannot simply say that if a node has an $n+1$'th child then the node has an n'th child. (HDLS) solves this problem by saying the immediate children of a node in an $n+1$-dimensional tree domain form an n-dimensional tree domain. In the two dimensional case, (HDLS) is thus the requirement that the children of a node in a tree form a list. (HDA) is a straightforward generalisation of the 2-dimensional ancestor property (A). The reader may wish to check that a one dimensional tree domain is a set of lists over 1 closed under prefixes, that is, T_1 is bijective to $\mathsf{List}(1)$. There are two zero dimensional tree domains which correspond to the empty tree and to the tree which just contains one node and no children.

The notion of automata is central in formal language theory and generalises to higher dimensions in a straightforward way. Firstly, we must extend tree domains so that higher dimensional trees can actually store data - this is done by associating to each path in a tree domain, a piece of data to be stored there.

Definition 2.4 (Labelled tree domains [11, Def 2.3]). *A Σ-labelled tree domain is a mapping $T \to \Sigma$, where T is a tree domain and Σ a set (called the alphabet). We denote the set of n-dimensional Σ-labelled tree domains by $T_n(\Sigma)$.*

Definition 2.5 (n-Automaton [11, Def 2.9]). *An $(n+1)$-dimensional automaton over an alphabet Σ and a finite set of states Q is a finite set of triples (σ, q, T) where $\sigma \in \Sigma$, $q \in Q$ and T is a Q-labelled tree domain of dimension n.*

Rogers goes on to define when an (n+1)-automaton licenses (or accepts) an $n+1$-dimensional tree as follows. A (Σ-labelled) local tree is an element of $\Sigma \times T_n(\Sigma)$. An (n+1)-dimensional grammar over Σ is a finite subset of $\Sigma \times T_n(\Sigma)$, ie a finite set of local trees. An element $\lambda : T \to \Sigma$ in $T_{n+1}(\Sigma)$ is licensed by a grammar if for all $s \in T$, the pair $(\lambda(s), \lambda' : T' \to \Sigma)$ is in the grammar, where $T' = \{w | s.w \in T\}$ and $\lambda'(w) = \lambda(s.w)$. In other words, a tree is licensed by a grammar if it is constructed from the local trees of the grammar. Note that, forgetting the alphabet Σ, an automaton can be seen as a grammar over Q. An element in $T_{n+1}(\Sigma)$ is now licensed by an automaton if it is an image of a Q-labelled tree licensed by the grammar in which the the label of the root of each local tree has been replaced with a symbol in Σ associated with that local tree in the automaton [11].

We will see in Section 5 that acceptance is more easily defined via the unique morphism from the initial algebra of trees. For coalgebraists let us note here already that automata are coalgebras. First, the notion of labelling means that n-dimensional tree domains form a functor $T_n : \mathsf{Set} \to \mathsf{Set}$. In particular $T_0(X) = 1 + X$ and $T_1(X) = \mathsf{List}(X)$. Now, an $n+1$-dimensional automata over Σ is just a finite set Q and a function $Q \to \mathcal{P}(\Sigma \times T_n(Q))$. Automata and their accepted languages will be discussed in detail in Section 5, but let us look at two familiar examples already.

Example 2.6. A 1-automaton is essentially the standard notion of a non-deterministic string automata — that is a function $Q \to \mathcal{P}(\Sigma \times (1 + Q))$ where each state can perform a Σ-transition and either terminate or arrive at another state.

Example 2.7. A 2-automaton is a coalgebra $\delta : Q \to \mathcal{P}(\Sigma \times T_1(Q))$, that is, a relation $\delta \subseteq Q \times (\Sigma \times \mathsf{List}(Q))$ which can be understood as a non-deterministic tree automata (see eg [6]): Given a state q and a tree $\sigma(t_1, \dots t_n)$ the automaton tries to recognise the tree by guessing a triple $(q, \sigma, [q_1, \dots q_n]) \in \delta$ and continuing this procedure in the states q_i with trees t_i. Whereas this coalgebraic definition has a top-down flavour, the accepted language is most easily defined in an algebraic (bottom-up) fashion as follows. The relation δ gives rise to a set of Q-labelled terms (or bottom-up *c*omputations) C via

$$\frac{(q, \sigma, []) \in \delta}{q\sigma \in C} \qquad \frac{(q, \sigma, [q_1, \dots q_n]) \in \delta, \quad q_i t_i \in C}{q\sigma(t_1, \dots t_n) \in C}.$$

where $q\sigma \in C$ means the automata recognises the σ-labelled tree starting from the state q. One then defines, wrt a set of accepting states Q_0, that the automaton accepts a tree t iff $qt \in C$ and $q \in Q_0$.

3 Higher Dimensional Trees, Algebraically

Despite being a natural generalisation of a 2-dimensional tree domain to an n-dimensional tree domain, Definition 2.3 is very concrete. For example, formalising the notion of licensing (following Definition 2.5 above) is tedious. We will show that a more abstract approach to the definition of tree domains is possible. In particular, the 1-dimensional tree domains are just the usual lists while the non-empty two-dimensional tree domains are known in the functional programming community as rose trees with a simple syntax and semantics. That is, categorically one may define $\mathsf{Rose}\, X = \mu Y. X \times \mathsf{List}\, Y$ and derive from this the equally simple Haskell implementation

```
data Rose a = Node a [Rose a]
```

What is really pleasant about this categorical/functional programming presentation of tree domains is that initial algebra semantics provides powerful methods for writing and reasoning about programs. In particular, it replaces fascination with the detailed representation of the structure of paths and the (LS) and (A) properties with the more abstract universal property of being an initial algebra. That is not to say paths are not important, just that they ought to be (in our opinion) a derived concept. Indeed, we show later in Theorem 4.6 how to derive the path algebra from the initial algebra semantics.

The natural question is whether we can give an initial algebra semantics for higher dimensional trees. The answer is not just yes, but yes in a surprisingly beautiful and elegant manner. As remarked earlier, the immediate children of a node in an $(n+1)$-dimensional tree should form an n-dimensional tree. This is formalised in

Definition 3.1. *Define a family of functors by*

$$\begin{array}{ll} R_{-1}X = 0 & T_n X = 1 + R_n X \quad (n \geq -1) \\ R_{n+1}X = \mu Y. X \times T_n Y & \end{array}$$

Note that we intend $R_{n+1}X$ to be the set of *non-empty* $n+1$-dimensional X-labelled tree domains while $T_{n+1}X$ is intended to be the set of *empty or non-empty* $n+1$-dimensional X-labelled tree domains. Thus $R_{n+1}X$ should consist of an element of X to be stored at the root of the tree and a potentially empty n-dimensional tree domain labelled with further tree domains. While one could start indexing at 0 by defining $R_0X = X$, there is no harm in starting one step before with the definition of -1-dimensional trees. As expected, calculations show that

n	R_nX	T_nX
-1	0	1
0	X	$1 + X$
1	$\mathsf{List}^+(X)$	$\mathsf{List}(X)$
2	$\mathsf{Rose}(X)$	$1 + \mathsf{Rose}(X)$

where $\mathsf{List}^+(X)$ are the non-empty lists over X.

In fact, one can go further and not just define a sequence of functors R_n and T_n, but a higher order functor which maps a functor F to the functor sending X to $\mu Y.X \times FY$. We find this particularly interesting for both theoretical and practical reasons. At the theoretical level, we note that this construction of a functor from a functor is the final piece of the jigsaw remarked upon in [9] and summarised in

	Monads	Comonads
Initial Algebras	$\mu Y. X + FY$	$\boldsymbol{\mu Y. X \times FY}$
Final Coalgebras	$\nu Y. X + FY$	$\nu Y. X \times FY$

In [9], the three other higher order functors were remarked upon as follows:

– The map sending a functor to F to the functor $X \mapsto \mu Y.X + FY$ is the free monad construction
– The map sending a functor to F to the functor $X \mapsto \nu Y.X + FY$ is the free completely iterative monad construction
– The map sending a functor to F to the functor $X \mapsto \nu Y.X \times FY$ is the cofree comonad construction

Higher dimensional tree functors provide—to our knowledge—the first naturally arising instance of the remaining quadrant of the table above. From [9], we have

Theorem 3.2. *For any functor F, the map $X \mapsto \mu Y.X \times FY$ is a comonad.*

At a practical level, this higher order functor translates into the following simple definition of higher dimensional trees in Haskell, the canonical recursion combinator arising from the initiality of higher dimensional trees and their comonadic structure. In the following, `Maybe` is Haskell implementation of the monad sending X to $1 + X$.

```
data Rose f a = Rose a (Maybe (f (Rose f a)))
type Tree f a = Maybe (Rose f a)

data Rose0 a = Rose0 a       -- = \X -> X
type Rose1   = Rose Rose0    -- = \X -> List^+(X)
type Rose2   = Rose Rose1    -- = \X -> Rose(X)
type Rose3   = Rose Rose2

cata :: Functor f => (a -> Maybe (f b) -> b) -> Rose f a -> b
cata g (Rose x xs) = g x (fmap (fmap (cata g)) xs)

instance Functor Rose0 where
  fmap f (Rose0 a) = Rose0 (f a)

instance Functor f => Functor (Rose f)
  where fmap f = cata act where act a t = Rose (f a) t

class Comonad f where
  root    :: f a -> a
  comult  :: f a -> f (f a)

instance Comonad Rose0 where
  root   (Rose0 x)   = x
  comult (Rose0 x)   = Rose0 (Rose0 x)

instance Functor f => Comonad (Rose f) where
  root (Rose x xs)   = x
  comult (Rose x xs) = Rose (Rose x xs) (fmap (fmap comult) xs)
```

As we have seen, higher dimensional trees are instances of canonical constructions which always produce comonads. It is also well-known that List^+ and List are monads. Less well known is that Rose is a monad. Clearly R_0 is also a monad. Indeed we have

Theorem 3.3. *For all $n \geq 0$, R_n is a monad.*

Space prevents us from detailing the proof of this theorem. However, it is important because it allows computation with higher dimensional trees to be further simplified via the use of the monadic notation available in Haskell to structure common patters of computation. For example, parsing and filtering become particularly simple.

To summarise, we depart from Rogers in not defining higher dimensional trees in terms of paths, but via the more abstract categorical notion of initial algebras. As a result, we take the *functor* T_n as primary as opposed to the *set* of tree domains which one may then label. This cleaner mathematical foundation reveals higher dimensional trees to be related to the fundamental constructions of the free monad, free completely iterative monad and cofree comonad. It also leads to a simple implementation of higher dimensional trees in Haskell.

4 Containers

Containers [8] are designed to represent those functors which are concrete data types and those natural transformations which are polymorphic functions between such concrete data types. Such data types include lists, trees etc, but not solutions of mixed variance recursive domain equations such as $\mu X.(X \to X) + \mathbb{N}$. Containers take as primitive the idea that concrete data types consist of its general form or *shapes* and, given such a shape, a set of *positions* where data can be stored. Since Rogers' n-dimensional trees certainly store data at the nodes of the n-dimensional tree, it is natural to ask whether these trees are indeed containers. In this section, we see that the functors T_n and R_n are indeed containers and point out the following theoretical and practical consequences:

- Many properties of n-dimensional trees can be deduced from the fact that they are containers. As just one example, our transformation of a non-deterministic automata into a deterministic one requires n-dimensional trees to preserve weak pullbacks. This follows from the fact that n-dimensional trees are containers.
- While we choose not to take paths and tree domains as primitive in our treatment of higher dimensional trees, paths are nevertheless important. We want a capability to compute with them but do not want the burden of verifying the (HDLS) and (HDA) properties. In particular, we want a purely inductive definition of tree domains and paths and, remarkably, find that the shapes and positions of the container T_n provide that.

Containers are semantically equivalent to *normal functors* and a special case of *analytic functors*. However, while containers talk about the different shapes a data structure can assume, analytic functors talk about the number of structures of a given size and hence there is no clear, simple and immediate connection between tree domains and paths on the one hand and analytic functors on the other hand. Thus we use containers rather than analytic functors to represent higher dimensional trees. In the rest of this section, we introduce containers and recall some of the closure properties of containers. This proves sufficient to then show that all Rogers' trees are indeed containers. While the theory of containers can be developed in any locally cartesian closed category with W-types and disjoint coproducts, we restrict to the category of Set to keep things simple.

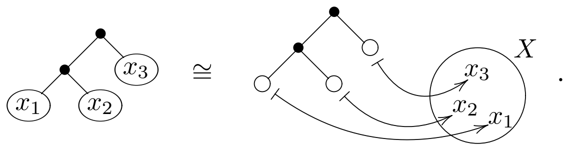
The simplest example of a data type which can be represented by a container is that of lists. Indeed, any element of the type $\mathsf{List}(X)$ of lists of X can be uniquely written as a natural number n given by the length of the list, together with a function $\{0, \ldots, n-1\} \to X$ which labels each position within the list with an element from X. Thus

$$\mathsf{List}(X) = \coprod_{n \in \mathbb{N}} \{0 \ldots n-1\} \to X$$

More generally, we consider data types given by i) *shapes* which describe the *form* of the data type; and ii) for each shape, $s \in S$, there is a set of positions $P(s)$. Thus we define

Definition 4.1 (Container). *A container (S, P) consists of a set S and an S-indexed family P of sets, ie a function $P : S \to \mathsf{Set}$.*

As suggested above, lists can be presented as a container with shapes $\mathbb{N}$ and positions defined by $P(n) = \{0, \ldots, n-1\}$. Similarly, any binary tree can be uniquely described by its underlying shape (which is obtained by deleting the data stored at the leaves) and a function mapping the positions in this shape to the data thus:



The extension of a container is an endofunctor defined as follows:

Definition 4.2 (Extension of a Container). *Let (S, P) be a container. Its extension, is the functor $T_{(S,P)}$ defined by*

$$T_{(S,P)}(X) = \coprod_{s \in S} P(s) \to X$$

Thus, an element of $T_{(S,P)}(X)$ is a pair (s, f) where $s \in S$ is a shape and $f : P(s) \to X$ is a labelling of the positions of s with elements from X. The action of $T_{(S,P)}$ on a morphism $g : X \to Y$ sends the element (s, f) to the element $(s, g \cdot f)$. If F is a functor that is the extension of a container, then the shapes of that container can simply be calculated as $F1$ — that is $S = T_{(S,P)}1$. This corresponds to erasing the data in a data structure to reveal the underlying shape. Containers have many good properties, in particular, many constructions on functors specialise to containers. These closure properties are summarised below

Theorem 4.3 (Closure properties of Containers [8]). *The following are true*

- *The identity functor is the extension of the container with one shape and one position.*
- *The constantly A valued functor has shapes A and positions given by $Pa = 0$.*
- *Let (S_1, P_1) and (S_2, P_2) be containers. Then the functor $T_{(S_1,P_1)} + T_{(S_2,P_2)}$ is the extension $T_{(S,P)}$ of the container (S, P) defined by*

$$S = S_1 + S_2 \qquad P(\mathsf{inl}(s)) = P_1 s \qquad P(\mathsf{inr}(s)) = P_2 s$$

- *Let (S_1, P_1) and (S_2, P_2) be containers. Then the functor $T_{(S_1,P_1)} \times T_{(S_2,P_2)}$ is the extension $T_{S,P}$ of the container (S, P) defined by*

$$S = S_1 \times S_2 \qquad P(s_1, s_2) = P_1 s_1 + P_2 s_2$$

In order to show that containers are closed under fixed points, we need to introduce the notion of a n-ary container to represent n-ary functors. For the purposes of our work, we only need bifunctors and so we restrict ourselves to binary containers

Definition 4.4 (Bi-Containers). *A bi-container consists of two containers with the same underlying shape. That is a set S and a pair of functions $P_1, P_2 : S \to \mathsf{Set}$. The extension of a binary container is a bifunctor given by*

$$T_{(S,P_1,P_2)}(X, Y) = \coprod_{s \in S} (P_1 s \to X) \times (P_2 s \to Y)$$

Given a bi-container (S, P_1, P_2), the functor $X \mapsto \mu Y.F(X, Y)$ is a container as demonstrated by the following theorem

Theorem 4.5 (Fixed Points of Containers [8]). *Let (S, P_1, P_2) be a bi-container and let $F(X, Y) = T_{(S,P_1,P_2)}(X, Y)$ be its extension. Then the functor $\mu Y.F(X, Y)$ is a container with shapes given by*

$$S = \mu Y.T_{(S,P_2)}(Y)$$

and positions given by

$$P(s, f) = P_1 s + \coprod_{p \in P_2 s} P(fp)$$

To understand this theorem, think of an element of $\mu Y.F(X, Y)$ as a tree with a top F-layer which stores elements from X at the X positions in this F-layer and further elements of $\mu Y.F(X, Y)$ at the Y-positions of this F-layer. We know that the shapes of the functor $\mu Y.F(X, Y)$ must be this functor at 1, ie $\mu Y.F(1, Y)$. More concretely, a shape for $\mu Y.F(X, Y)$ must thus be an F-shape for the top layer of a tree and, for each Y-position of that shape, we must have a shape of $\mu Y.F(X, Y)$ to represent the tree recursively stored at that position. As for the positions for storing data of type X in a tree with shape (s, f) where $s \in S$ and $f : P_2 s \to \mu Y.F1Y$, these should be either the positions for storing X-data in the top layer given by $P_1 s$ or, for each position in $p \in P_2 s$, a position in the subtree stored at that position. Since that subtree has shape fp, we end up with the formula above.

Applying these closure properties, we derive the following

Theorem 4.6. *Rogers' n-dimensional non-empty tree functor R_n is the extension of a container. That is, $R_n = T_{(S^+_n, P^+_n)}$ where*

$$S^+_{-1} = 0$$

$$\begin{aligned} S^+_{n+1} &= \mu Y.1 + R_n Y \\ P^+_{n+1}(\mathsf{inl}*) &= 1 \\ P^+_{n+1}(\mathsf{inr}(s, f)) &= 1 + \coprod_{p \in P_n s} P^+_{n+1}(fp) \end{aligned}$$

As a corollary, T_n is also the extension of a container. That is $T_n = T_{(S_n, P_n)}$ where $S_n = 1 + S^+_n$, $P_n(\mathsf{inl}*) = 0$ and $P_n(\mathsf{inr}s) = P^+_n s$. What is particularly nice about the container presentation of T_n is that the shapes S_n are in bijection with the tree domains while the paths in any tree domain are in bijection with the positions of the equivalent shape. Further, the paths are given by a purely inductive definition.

5 Automata, (Co)algebraically

We show that the classical automata-theoretic results about determinisation and minimisation extend to the higher-dimensional automata of Rogers. Using our reformulation of Rogers' structures in Section 3 and the container-technology of Section 4, these results become special cases of the classical results about automata as algebras for a functor, a theory initiated by Arbib and Manes [2–4]. We also extend Rogers' work by appropriate notions of signature and deterministic automata.

We should like to point out that none of the constructions or proofs in this section requires the explicit manipulation of trees or tree domains.

Before starting on the topic of the section, we review the situation for string and tree automata. Ignoring initial and accepting states, the situation is depicted in

(strings)	non-det	det
top-down	$Q \to \mathcal{P}(A \times Q)$	$Q \to Q^A$
bottom-up	$A \times Q \to \mathcal{P}Q$	$A \times Q \to Q$

(trees)	non-det	det
top-down	$Q \to \mathcal{P}(FQ)$	—
bottom-up	$FQ \to \mathcal{P}Q$	$FQ \to Q$

For both string and tree automata, the relationship between non-deterministic top-down automata (=coalgebras in the Kleisli-category of $\mathcal{P}$) and non-deterministic bottom-up automata (=algebras in the Kleisli-category of $\mathcal{P}$) is straightforward: both $Q \to \mathcal{P}(FQ)$ and $FQ \to \mathcal{P}Q$ are just two different ways of denoting a relation $\subseteq Q \times FQ$. The relationship between deterministic top-down automata (=coalgebras) and deterministic bottom-up automata (=algebras) is given in the string case by the adjunction $A \times - \dashv (-)^A$ (this situation is generalised and studied in [3]). In the tree case, F is an arbitrary functor on Set and so has in general no right-adjoint.[1] It is still possible to describe deterministic top-down tree automata but they are strictly less expressive [6, Chapter 1.6].

The familiar move from non-deterministic to deterministic string automata can be summarised as follows. Any non-deterministic transition structure $f : Q \to \mathcal{P}(A \times Q)$ can be lifted to a map $\bar{f} : \mathcal{P}Q \to \mathcal{P}(A \times Q)$ given by $\bar{f}(S) = \bigcup_{q \in S} f(q)$. Using $\mathcal{P}(A \times Q) \cong (\mathcal{P}Q)^A$, $\bar{f}$ is a deterministic transition structure $\mathcal{P}Q \to (\mathcal{P}Q)^A$ on $\mathcal{P}Q$. Determinisation for tree automata will be discussed below.

5.1 Signatures

Rogers' automata of Definition 2.5 do not associate arities to the symbols in the alphabet Σ. For example, in the tree automata of Example 2.6, one σ may appear in two triples $(q, \sigma, l_1), (q, \sigma, l_2) \in \delta$ where l_1 and l_2 are lists of different lengths. Thus the same 'function symbol' σ may have different arities and the Σ-labelled trees are not exactly elements of a term algebra.

To rectify this situation, we must ask ourselves what is the appropriate notion of arity if operations take as input higher dimensional trees. In the two-dimensional case arities are natural numbers: the arity of a function symbol σ is the number of its arguments. But, in container terminology, $\mathbb{N}$ is just the the set of shapes of $T_1 = \mathsf{List}$. Thus, when operations of a signature are consuming higher dimensional trees, their arities should be the shapes of trees one dimension lower. This leads to

Definition 5.1 ($(n+1)$-dimensional signature). *An $(n+1)$-dimensional signature is a set Σ with a map $\Sigma \to T_n(1)$.*

Example 5.2. 1. A 1-dimensional signature is a map $r : \Sigma \to \{0, 1\}$, due to the isomorphism $T_0(1) \cong \{0, 1\}$. We will see below (Example 5.4) that 0 specifies nullary operations and 1 specifies unary operations.
2. A 2-dimensional signature is a signature in the usual sense, due to the isomorphism $T_1(1) \cong \mathbb{N}$ that maps a list to its length.

The next step is to associate to each signature a functor in such a way that the initial algebra for the functor contains the elements of the language accepted by an automaton. The simplest and most elegant way to do this is to construct a container and use its extension. Recalling that $P_n : S_n \to \mathsf{Set}$ is the container whose extension is T_n and that $S_n = T_n(1)$, we can turn any signature $r : \Sigma \to T_n(1)$ into the container (Σ, P_r) as follows

$$P_r : \Sigma \xrightarrow{r} S_n \xrightarrow{P_n} \mathsf{Set}. \tag{1}$$

Definition 5.3 (F_Σ). *The functor $F_{(\Sigma,r)}$, or briefly F_Σ, associated to a signature $r : \Sigma \to T_n(1)$ is the extension $T_{(\Sigma,P_r)}$ of the container (1), that is, $F_\Sigma(X) = \coprod_{\sigma \in \Sigma} P_n(r(\sigma)) \to X$.*

Example 5.4. 1. A 1-dimensional signature $r : \Sigma \to \{0, 1\}$ gives rise to the functor $F_\Sigma(X) = \Sigma_0 + \Sigma_1 \times X$ where $\Sigma_i = r^{-1}(i)$.
2. A 2-dimensional signature $r : \Sigma \to \mathbb{N}$ gives rise to the functor $F_\Sigma(X) = \coprod_{\sigma \in \Sigma} X^{r(\sigma)}$ usually associated with a signature.

[1] In fact, if $F : \mathsf{Set} \to \mathsf{Set}$ has a right-adjoint, then $F = A \times -$ for $A = F1$.

The next two propositions, which one might skip as a pedantic technical interlude, make the relation between an alphabet Σ' and a signature $\Sigma \to T_n(1)$ precise. The first proposition says that trees for the signature $\Sigma \to T_n(1)$ (ie elements of the initial F_Σ-algebra) are also trees over the alphabet Σ (ie elements of $T_{n+1}(\Sigma)$). The second proposition says that trees over the alphabet Σ' are the same as trees over the signature $\Sigma' \times T_n(1) \to T_n(1)$.

Proposition 5.5. *For each $(n+1)$-dimensional signature $\Sigma \to T_n(1)$, there is a canonical F_Σ-algebra structure on $T_{n+1}(\Sigma)$. Moreover, the unique algebra morphism from the initial F_Σ-algebra to $T_{n+1}(\Sigma)$ is injective.*

Proof. The carrier of the initial F_Σ-algebra is $\mu Y.F_\Sigma(Y)$ and $R_{n+1}(\Sigma)$ is $\mu Y.\Sigma \times T_n(Y)$. The injective morphism in question arises from the injective map of type $F_\Sigma(Y) \to \Sigma \times T_n(Y)$, that is of type $(\coprod_{\sigma\in\Sigma} P(r(\sigma)) \to Y) \to \Sigma \times (\coprod_{s\in S_n} P(s) \to Y)$, which maps pairs $(\sigma, f) \in F_\Sigma(Y)$ to $(\sigma, (r(\sigma), f))$.

Proposition 5.6. *Let Σ' be a set (called an alphabet) and Σ be the signature given by the projection $r : \Sigma' \times T_n(1) \to T_n(1)$. Then $R_{n+1}(\Sigma')$ is isomorphic to the (carrier of the) initial F_Σ-algebra.*

Proof. The carrier of the initial F_Σ-algebra is $\mu Y.F_\Sigma(Y)$ and $R_{n+1}(\Sigma')$ is $\mu Y.\Sigma' \times T_n(Y)$. But $\Sigma' \times T_n(Y) = \Sigma' \times (\coprod_{s\in S_n} P(s) \to Y) \cong \coprod_{(\sigma,s)\in\Sigma'\times S_n} P(s) \to Y = \coprod_{\sigma\in\Sigma} P(r(\sigma)) \to Y = F_\Sigma(Y)$.

5.2 Higher Dimensional Automata

Before giving a coalgebraic formulation of Rogers' automata (Definition 5.10), we introduce the corresponding notion of deterministic automaton (Definition 5.7), which has a particularly simple definition of accepted language and is used in the next subsection on determinisation and minimisation. (Recall Definition 5.3 of F_Σ.)

Definition 5.7. *A deterministic $(n+1)$-dimensional automaton for the signature $\Sigma \to T_n(1)$ is a function*

$$F_\Sigma Q \to Q.$$

Example 5.8. 1. To obtain the usual string automata over an alphabet A we consider a 1-dimensional signature Σ consisting of the elements of A as unary operation symbols plus one additional nullary operation symbol (see Example 5.4.1). $F_\Sigma(Q)$ is then $1 + A \times Q$.
2. A 2-dimensional automaton is the usual deterministic bottom-up tree automaton [6].

Definition 5.9. *A state q in a deterministic $(n+1)$-dimensional automaton for the signature $\Sigma \to T_n(1)$ accepts an $(n+1)$-dimensional tree t if the unique morphism from the initial F_Σ-algebra maps t to q.*

We adapt Rogers' definition of automata given in Definition 2.5:

Definition 5.10. *An $(n+1)$-dimensional automaton for the signature $\Sigma \to T_n(1)$ is a function*

$$Q \to \mathcal{P}(F_\Sigma(Q)).$$

Example 5.11. 1. In the case of string automata, $F_\Sigma(Q)$ is $1 + A \times Q$ and an automaton becomes $Q \to \mathcal{P}(1 + A \times Q) \cong 2 \times (\mathcal{P}Q)^A$. The map $Q \to 2$ encodes the accepting states and the map $Q \to (\mathcal{P}Q)^A$ gives the transition structure.

2. Comparing with the previous definition, a 2-dimensional automaton $\delta : Q \to \mathcal{P}(F_\Sigma(Q))$ can still be considered as a set of triples $\delta \subseteq Q \times (\Sigma \times \mathsf{List}(Q))$, but not all such triples are allowed: for $(q, \sigma, \langle q_1, \dots q_n \rangle) \in \delta$ it has to be the the case that the arity of σ is n. This coincides with the notion of a non-deterministic top-down tree automaton as in [6].

We have indicated how to define the accepted language of a (non-deterministic) automaton in Example 2.7. In particular, we found it natural to give a bottom-up formulation. We will now generalise this definition. The basic idea is as follows. We first observe that we cannot use the final coalgebra for the functor $\mathcal{P}F_\Sigma$ since this coalgebra would take the branching given by $\mathcal{P}$ into account. Instead, the correct idea is to consider a non-deterministic automaton as a F_Σ-coalgebra in the category of relations. We first note the following proposition which follows from F_Σ being the extension of a container.

Proposition 5.12. *F_Σ preserves weak pullbacks.*

Now let Rel denote the category of sets and relations.

Definition 5.13. *Given a functor F on Set we define $\bar{F}$ to map sets X to $\bar{F}X = FX$ and to map relations $X \overset{\pi_0}{\leftarrow} R \overset{\pi_1}{\to} Y$ to $\bar{F}R = F(\pi_0)^\circ ; F(\pi_1)$ where $(-)^\circ$ denotes relational converse and ';' relational composition .*

Barr [5] showed that $\bar{F}$ is a *functor* on Rel if and only if F preserves weak pullbacks. A theorem of de Moor [7, Theorem 5] and Hasuo et al [10, Theorem 3.1] then guarantees that the initial F-algebra $i : FI \to I$ in Set gives rise to the final $\bar{F}$-coalgebra $i^\circ : I \to \bar{F}I$ in Rel. This gives a 'coinductive' definition of the accepted language of a non-deterministic automaton:

Definition 5.14. *The language accepted by a state q of an $(n+1)$-dimensional automaton $Q \to \mathcal{P}(F_\Sigma(Q))$ is given by the unique arrow (in the category Rel) into the final $\bar{F}_\Sigma$-coalgebra .*

Note that this definition associates to q a subset of the carrier I of the initial F_Σ-algebra.

It is clear from the constructions that every deterministic automaton can be considered as a non-deterministic automaton, and that the two notions of accepted language agree. We make this precise with the following definition and proposition.

Definition 5.15. *The non-deterministic automaton corresponding to the deterministic automaton $f : F_\Sigma Q \to Q$ is given by $f^\circ : Q \to \mathcal{P}F_\Sigma Q$ (where f° is again the converse relation of (the graph of) f).*

Proposition 5.16. *The deterministic automaton $F_\Sigma Q \to Q$ accepts t in q if and only if the corresponding non-deterministic automaton $Q \to \mathcal{P}F_\Sigma Q$ has t in the language of q.*

5.3 Determinisation and Minimisation

This section follows the work by Arbib and Manes [2–4] on automata as algebras for a functor on a category.

Determinisation First observe that the elementship relation $\ni \subseteq \mathcal{P}X \times X$ can be lifted to $\bar{F}(\ni) \subseteq F\mathcal{P}X \times FX$, which can be written as

$$F\mathcal{P}X \xrightarrow{\tau_X} \mathcal{P}FX \tag{2}$$

τ_X is well-known to be natural in X whenever F preserves weak pullbacks. Now, given a non-deterministic automaton

$$Q \to \mathcal{P}F_\Sigma Q \tag{3}$$

we first turn it from top-down to bottom-up by going to the converse relation

$$F_\Sigma Q \to \mathcal{P}Q \tag{4}$$

and then lift it from $F_\Sigma Q$ to $\mathcal{P}F_\Sigma Q$ and precompose with τ to obtain

$$F_\Sigma \mathcal{P}Q \to \mathcal{P}F_\Sigma Q \to \mathcal{P}Q \tag{5}$$

Remark 5.17. The step from (4) to (5) is a special case of [4, Lemma 7] (where $\mathcal{P}$ can be an arbitrary monad on a base category).

Theorem 5.18. *Given an $(n+1)$-dimensional automaton $Q \to \mathcal{P}F_\Sigma Q$ (Definition 5.10) with accepting states $Q_0 \subseteq Q$, the state Q_0 in the corresponding deterministic automaton (5) accepts the same language.*

Minimisation A deterministic automaton with a set of accepting states is a structure

$$F_\Sigma Q \xrightarrow{\delta} Q \xrightarrow{\alpha} 2 \tag{6}$$

We denote by $F_\Sigma I \to I$ the initial F_Σ-algebra and by $\rho : I \to Q$ the unique morphism given by initiality. The map $\beta = \alpha \circ \rho$ is called the *behaviour* of (6) because $\beta(t)$ tells us for any $t \in I$ whether it belongs to the accepted language or not. Note that the automata (6) form a category, denoted DAut, which has as morphism $f : (\delta, \alpha) \to (\delta', \alpha')$ those algebra morphism $f : \delta \to \delta'$ satisfying $\alpha' \circ f = \alpha$.

Definition 5.19 ([2, Section 4]). *Let $\iota : F_\Sigma I \to I$ be the initial F_Σ-algebra. The automaton (6) is* reachable *if the algebra morphism $\iota \to \delta$ is surjective and it is a* realisation *of $\beta : I \to 2$ iff there is a morphism $(\iota, \beta) \to (\delta, \alpha)$ in* DAut. *Moreover, (6) is a minimal realisation of β iff for all reachable realisations (δ', α') of β there is a unique surjective* DAut*-morphism $f : (\delta', \alpha') \to (\delta, \alpha)$.*

Different minimal realisation theorems can be found in Arbib and Manes [2–4] and Adámek and Trnková [1]. The theorem below follows [1, V.1.3].

Theorem 5.20. *Let Σ be an $(n+1)$-dimensional signature, F_Σ the corresponding functor and $F_\Sigma I \to I$ the initial F_Σ-algebra. Then every map $\beta : I \to 2$ has a minimal realisation.*

Proof. Let $e_i : (\iota, \beta) \to (\delta_i, \alpha_i)$ be the collection of all surjective DAut-morphisms with domain (ι, β). Let f_i be the multiple pushout of e_i in Set and $g = f_i \circ e_i$. The universal property gives us α with $\alpha \circ g = \beta$. Being a container F_Σ is finitary and, therefore [1, V.1.5], preserves the multiple pushout. Hence there is δ with $\delta \circ F_\Sigma g = g \circ \iota$. Since F_Σ preserves, like any set-functor, surjective maps, δ is uniquely determined. We have constructed an automaton (δ, α) that realises β. It is minimal because any other reachable realisation appears as one of the e_i.

6 Conclusion

This paper applies (co)algebraic and categorical techniques to Rogers' recent work in linguistics on higher dimensional trees. In particular, we have given an algebraic formulation of Rogers' higher dimensional trees and automata. Our analysis shows that, just as ordinary trees, the higher dimensional trees organise themselves in an initial algebra for a set-functor. This allowed us to use Arbib and Manes' theory of automata as algebras for a functor, yielding simple definitions of accepted language and straightforward constructions of determinisation and minimisation.

More importantly, as we have only been able to hint at, our algebraic formulation gives us the possibility to write programs manipulating the trees in functional programming languages like Haskell that support polymorphic algebraic data types. Future work will be needed to substantiate our claim that, in fact, our abstract categorical treatment is very concrete in the sense that it will give rise to simple implementations of algorithms manipulation higher dimensional trees. A good starting point could be Rogers' characterisation of non-strict tree adjoining grammars as 3-dimensional automata [11, Thm 5.2].

References

1. J. Adámek and V. Trnková. *Automata and Algebras in Categories*. Kluwer, 1990.
2. M. A. Arbib and E. G. Manes. Machines in a category: An expository introduction. *SIAM Review*, 16, 1974.
3. M. A. Arbib and E. G. Manes. Adjoint machines, state-behaviour machines, and duality. *Journ. of Pure and Applied Algebra*, 6, 1975.
4. M. A. Arbib and E. G. Manes. Fuzzy machines in a category. *Bull. Austral. Math. Soc.*, 13, 1975.
5. M. Barr. Relational algebras. *LNM*, 137, 1970.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997. Available on: `http://www.grappa.univ-lille3.fr/tata`.
7. O. de Moor. Inductive data types for predicate transformers. *Information Processing Letters*, 43(3):113–118, 1992.
8. N. Ghani, M. Abbott, and T. Altenkirch. Containers - constructing strictly positive types. *Theoretical Computer Science*, 341(1):3–27, 2005.
9. N. Ghani, C. Lüth, F. de Marchi, and J. Power. Dualizing initial algebras. *Mathematical Structures in Computer Science*, 13(1):349–370, 2003.
10. I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace theory. In *International Workshop on Coalgebraic Methods in Computer Science (CMCS 2006)*, volume 164 of *Elect. Notes in Theor. Comp. Sci.*, pages 47–65. Elsevier, 2006.
11. J. Rogers. Syntactic structures as multi-dimensional trees. *Research on Language and Computation*, 1(3-4):265–305, 2003.