Snoc Trees: Growing Trees From the Bottom Up

Neil Ghani
Kodamai Ltd
neil@kodamai.com

Fredrik Nordvall Forsberg
University of Strathclyde
fredrik.nordvall-forsberg@strath.ac.uk

Samuel Fish
University of Strathclyde
samuel.fish@strath.ac.uk

We introduce snoc-trees which generalise snoc-lists, i.e., backwards lists where new elements are added at the back rather than at the front of the list. We uncover the categorical and computational structure of snoc-trees using both functorial and container-theoretic approaches, and show the power of their associated fold operator. We relate snoc-trees to the usual trees arising from initial algebra semantics, and demonstrate the use of snoc-trees via an example arising from a type theoretic variant of compositional sampling from a probability distribution.

1 Introduction

Trees are a ubiquitous in computer science. Their formal definition can be given set theoretically (e.g. using tree domains) but this is a poor approach due to low levels of abstraction. A much better approach — widely adopted in functional programming, type theory and category theory — is to define trees via *initial algebra semantics* [11]. This offers a formal denotational semantics for inductive data types as the initial algebra of a functor; from this single concept one can derive introduction rules, elimination rules, computation rules, and extensionality rules. The theory is also robust, scaling to more complex forms of data structures including mutually recursive data types, inductive families, and inductive-recursive types, among others. Applications in functional programming include fold combinators, Church encodings, build combinators and shortcut fusion rules. Initial algebra semantics also has a dual final coalgebra semantics, which is widely used across computer science to model e.g. dynamical systems with infinitary behaviour [17]. Not bad for a tiny bit of category theory!

However, consider the following problem which highlighted to us the advantage of snoc-trees: how do we create a universe of types all of whom support sampling? Here, sampling means a function that can produce elements of a type according to a given probability distribution [5]. Our goal — as developed in Section 5 — is to lift sampling from basic types such as Bool to arbitrary types in a universe including for example coproducts, products, and inductive types. As we shall see, in lifting sampling to inductive types, we naturally encounter the need for snoc-trees.

To be more precise, let $\mathsf{D} : \mathsf{Set} \to \mathsf{Set}$ be the (finite support) distributions monad [12], and let $\mathsf{R} : \mathsf{Set} \to \mathsf{Set}$ be a monad encapsulating random effects [15]. To sample from a type A means to implement a function $f : \mathsf{D}A \to \mathsf{R}A$ which produces random values $f(\phi)$ of type A according to its input distribution ϕ. We assume that we are given samplers for basic types such as Bool, and would like to compositionally build samplers for other types in the universe using induction on type structure, as is typical in generic programming. As a specific example, lets say we can sample from the type A, i.e., we have a function of type $\mathsf{D}A \to \mathsf{R}A$, and we want to build a function to sample over lists containing data of type A, i.e.,

Submitted to:

we want to build a function of type $\mathsf{D}(\mathsf{List}A) \to \mathsf{R}(\mathsf{List}A)$. The algorithm we are going to look at uses an accumulator of the list already built, as demonstrated by the following Agda [4] implementation — see Section 5 for a full explanation of the code, but for now, the reader need only understand the overall structure:

```
sampleList : {A : Set} → (D A → R A) → List A → D (List A) → R (List A)
sampleList sampleA xs ϕ = do
  r <- sampleMaybe sampleA (mapD head ϕ)
  case r of λ where
    nothing → return xs
    (just a) → sampleList sampleA (xs ++ [ a ]) (reindexD (a ∷_) ϕ)
```

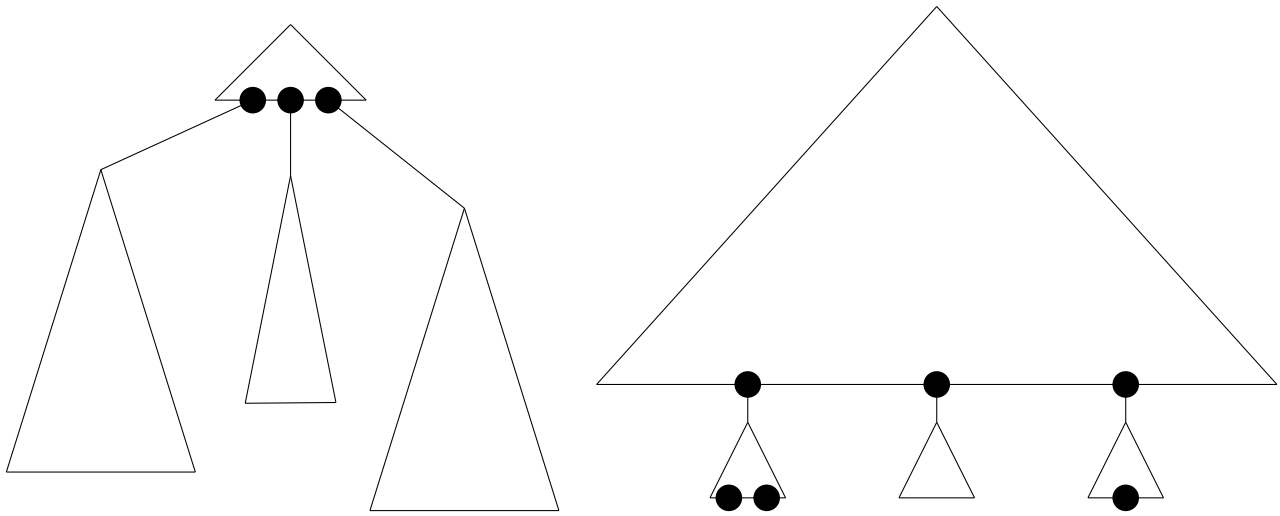
It is far from obvious why this program terminates (and indeed Agda cannot see it), but note that this paper is not about the correctness of this or any other sampling algorithm mentioned — that is the subject of work currently being written up. Rather this paper is about the data structures needed by sampling.

The algorithm works as follows: at an informal level, we have already built the initial segment of the output list `xs`. We construct from the distribution over lists ϕ a new distribution over whether we should continue to build the list or just terminate and output the list `xs`. If we are to continue, we get the next element `a`, add it to the accumulator, and continue to build the rest of the output list. That involves creating a new distribution to continue to sample from. Crucially, this is *not* how we build lists usually, where constructors add an element to the front of a list — rather we are adding an element to the end of a list. Thus the algorithm is really building snoc-lists. Of course, snoc-lists are isomorphic to lists, so there is no real problem from either theoretical or practical perspectives.

However, the problem becomes much more acute for more general inductive structures such as trees, which involve branching. From a practical perspective, if we had a distribution ϕ over trees, how we sample the left subtree and how we sample the right subtree are not independent. For example, the distribution ϕ might have no probability mass if the left and right subtrees do not have the same size. Thus we need to recursively sample not from a tree, but from a family of trees representing a joint distribution over potential subtrees. Further, if the accumulator is the partially constructed tree, then we build the tree not by creating a new root, but by adding to the growth points at the bottom of the partially constructed tree. This naturally creates not normal tress but rather what we will call snoc-tres.

Figure 1 shows the intuitive difference between normal trees and snoc-trees. The former consist of a constructor with all growth points filled in recursively by subtrees. The later consist of a subtree with all growth points filled in by constructors. This paper shows that all the goodies one gets from initial algebra semantics actually apply to snoc-trees as well as normal trees because initial algebra semantics is robust enough to cover snoc-trees. One must just deploy initial algebra semantics in a different category!

Related work This paper is fundamentally about categorical approaches to data structures and how initial algebra semantics can be used to model them. Initial algebra semantics was pioneered in [11] and has since become the cornerstone of the categorical approach to data types. Of particular interest in this paper is the extension of initial algebra semantics to higher kinded/ranked data structures [2, 13]. We have previously used snoc-trees in the study of data types for continuous functions [10] but their presentation via inductive-recursive definitions [8] was all that was given. This paper, in contrast, develops their



(a) A regular tree with subtrees (b) A snoc-tree

Figure 1: Regular trees versus snoc-trees

fundamental theory and their motivation as a foundation for type-driven sampling. Containers were introduced as a foundation for concrete data types in [1].

2 Initial Algebra Semantics

The essence of initial algebra semantics is the following: Given an endofunctor $F : \mathbb{C} \to \mathbb{C}$ on a category $\mathbb{C}$, we construct the category of F-algebras, denoted F-Alg, whose objects are pairs (X, g) where X is an object of $\mathbb{C}$ and $g : FX \to X$. A morphism (X, g) to (Y, h) is a map $\alpha : X \to Y$ in $\mathbb{C}$ such that $\alpha \circ g = h \circ F\alpha$. The initial object of F-Alg is called the initial F-algebra and is denoted μF.

Example 2.1. *Let $FX = 1 + A \times X$ for some object A of Set. Then μF is the datatype of lists whose elements are of type A. If $FX = 1 + X \times X$, then μF is the data type of binary tress with no data at their leaves.*

The existence of initial algebras is well studied, and there are mild conditions guaranteeing plenty of initial algebras, e.g. initial algebras exist if the underlying category has an initial object and ω-colimits, which are preserved by the functor [3]. In the rest of this paper we assume initial algebras to exist as needed. Note that if $F : \mathbb{C} \to \mathbb{C}$ is a functor, we can consider the functor T_F which sends X to the initial algebra of the functor $X + F$. As is well known, T_F is the free monad on F. Indeed, T_F is itself an initial algebra of the functor $\mathsf{Id} + F \circ - : [\mathbb{C}, \mathbb{C}] \to [\mathbb{C}, \mathbb{C}]$, that is,

$$T_F = \mu K.\mathsf{Id} + F \circ K \ .$$

This definition of T is an example of a rank 2 data type [2]. Intuitively, $T_F(X)$ are terms of the inductive type specified by F with variables drawn from X. Since variables can be substituted for by further trees, locations of variables in a tree will be described in this paper as growth points — this distinguishes such locations from locations of nullary constructors where no growth is possible. As $\mu F = T_F\emptyset$, we can see

that the initial algebra of a functor has no growth points. The fundamental intuitive difference between normal trees and snoc-trees (see Figure 1) can now be described:

- Normal trees consist of a constructor, each of whose growth points is mapped to another normal tree;
- Snoc-trees consist of a sub-snoc tree, each of whose growth points is mapped to a constructor (each of which of course have their own growth points).

This observation will be formalised in Sections 3 and 4, where we describe normal trees and snoc-trees via functors and then via containers. For now, we note the primacy within snoc-trees of growth points means their formal definition is given by a rank-2 definition [2]. This is the first of several fundamental differences between normal trees and snoc-trees.

3 A Functorial Formalisation of Snoc-Trees

In this section we give a functorial presentation of snoc-trees. We give what we believe is the most natural definition, namely one using initial algebras of functors over functor categories. This definition is the cleanest and most abstract we have found, transparently shows the relationship with normal trees (and calculi of explicit substitutions), and can be used to establish basic results such as the embedding of snoc-trees as normal trees. We extract the associated elimination operator and show its use. We also provide an alternative functorial definition, and show it equivalent to our initial algebra definition.

3.1 A Rank 2 Initial Algebra Semantics for Snoc-Trees

We define snoc-trees as follows, together with the free monad for comparison:

Definition 3.1. *Let $F : \mathbb{C} \to \mathbb{C}$ be a functor. We denote by μF the initial F-algebra, by F^* the free monad on F, and by F_* the initial algebra of the functor $\mathsf{Id} + - \circ F : [\mathbb{C},\mathbb{C}] \to [\mathbb{C},\mathbb{C}]$. That is*

$$F^* = \mu K.\mathsf{Id} + F \circ K \quad \textit{and} \quad F_* = \mu K.\mathsf{Id} + K \circ F$$

To understand both these definitions F^* and F_*, note that both offer the possibility of terminating the recursion via the Id constructor. If X is thought of as an object of variables, this means that each variable is both a normal tree and a snoc-tree. The recursive case has, for normal trees F^*, the functor F after the recursive call [1] meaning the constructors from F form the root of the tree, and all of the growth points of F are attached to other normal trees. For snoc-trees F_*, the reverse is the case, meaning that the root is given by a snoc-tree, and each of the growth points of that snoc-tree is attached to a constructor from F. This matches both the intuitive description in Section 2 and Figure 1. We note in passing the similar formula $\mu K.\mathsf{Id} + F + K \circ K$ representing a calculus of terms of F with an explicit substitutions operator.

We can immediately prove one intuitive relationship between snoc-trees and regular trees using the universal property of snoc-trees:

[1] Recall $F \circ K$ says first apply K and then apply F.

Lemma 3.2. *There is a function from snoc-trees to ordinary trees which preserve the labels of the trees.*

Proof. Build a $\mathsf{Id} + - \circ F$ algebra for F^*. The unit of F^* gives one of the two maps required. The other is the map $F^* \circ F \to F * \circ F^* \to F^*$ where the first map is derived from the freeness of F^* over F, and the second is the multiplication of the monad F^*. □

At this stage, it might be tempting to think that normal trees and snoc-trees are isomorphic, generalising the situation between normal lists and snoc-lists. This is not the case, because snoc-trees have all growth points at the same depth from the root. This is formalised in Lemma 3.3 below. We finish this section by considering the elimination rule for snoc-trees — what functional programmers would call the associated fold-rule. If we write $\mathsf{Nat}(F,G)$ for the natural transformations between functors F and G, pure initial algebra semantics gives the eliminator

$$\mathsf{fold}_G : \mathsf{Nat}(\mathsf{Id} + G \circ F, G) \to \mathsf{Nat}(F_*, G)$$

for every $G : \mathsf{Set} \to \mathsf{Set}$. At first sight, this seems limited — what does one do if one wants to define a function, not a natural transformation? For example, given the functor $FX = 1 + X \times X$ on Set, one might want to do something simple like $\mathsf{sum} : F_*(\mathsf{Int}) \to \mathsf{Int}$. The trick is to notice that this is indeed natural for the right choice of functor: sum is a natural transformation from $F_* K_{\mathsf{Int}}$ to K_{Int}, where K_{Int} is the constantly Int-valued functor. For the example of sum, such a natural transformation is easy to give. In general, more sophisticated examples require the use of Mendler Algebras or, categorically, right Kan extensions [2, 13], which use initial algebra semantics to create a generalised fold operator for defining natural transformations $\mu F \circ G \to H$.

3.2 Alternative presentations of snoc-trees

Assuming $\mathbb{C}$ has (countable) coproducts, we can also give a concrete construction of F_* as the coproduct $\sum_{n\in\mathbb{N}} F^n$, i.e., as the functor $\mathbb{C} \to \mathbb{C}$ which sends X to $\sum_{n\in\mathbb{N}} (\underbrace{F \circ \ldots \circ F}_{n \text{ times}})(X)$.

Lemma 3.3. *$\sum_{n\in\mathbb{N}} F^n$ is the initial $(\mathsf{Id} + \_ \circ F)$-algebra; that is, $F_* = \sum_{n\in\mathbb{N}} F^n$.*

Proof. The algebra structure map $\mathsf{Id} + (\sum_{n\in\mathbb{N}} F^n) \circ F \to \sum_{n\in\mathbb{N}} F^n$ is induced by the coproduct injections $\mathsf{inj}_0 : X \to \sum_{n\in\mathbb{N}} F^n(X)$ and $\mathsf{inj}_{n+1} : F^n(F(X)) \to \sum_{n\in\mathbb{N}} F^n(X)$. Given an algebra $[\sigma, \rho] : \mathsf{Id} + H \circ F \to H$, we define a family of natural transformations $\theta_n : F^n \to H$ by recursion on n: we set $\theta_0 = \sigma$, and $\theta_{n+1,X} = \rho_X \circ \theta_{n,FX}$. This makes up a natural transformation $\sum_{n\in\mathbb{N}} F^n \to H$, which can be proven to be unique by induction on n and the universal property of the coproduct. □

In general, initial algebras can be computed as a colimit; it is perhaps noteworthy that F_* can be computed simply as a coproduct instead. It is also natural to ask if F_* can be characterised further beyond being an initial algebra. It turns out that while F^* is the free monad on F, F_* is the free $\mathbb{N}$-*graded* monad on F.

Definition 3.4. *Let $(E, \cdot, 1)$ be a monoid An E-*graded monad *T on a category $\mathbb{C}$ consists of functions*

$$T_e : |\mathbb{C}| \to |\mathbb{C}|,$$

for each $e \in E$, equipped with a unit *transformation η and* Kleisli extension *operator $(\_)^\dagger$ such that*

$$\eta_X : X \to T_1 X$$
$$f_d^\dagger : T_d X \to T_{d \cdot e} Y$$

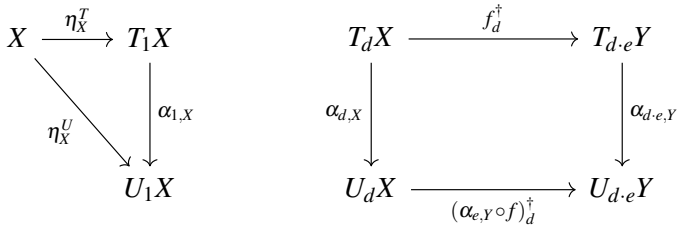
for $f : X \to T_e Y$. These operations are required to satisfy the following laws

$$f_1^\dagger \circ \eta_X = f$$
$$(\eta_X)_e^\dagger = \mathsf{id}_{T_e X}$$
$$\left(g_e^\dagger \circ f\right)_d^\dagger = g_{d \cdot e}^\dagger \circ f_d^\dagger,$$

for any $g : Y \to T_{e'} Z$.

In particular, we see that a $\mathbf{1}$-graded monad, where $\mathbf{1}$ is the trivial monoid with one element, is nothing but an ordinary monad (in Kleisli triple form). We extend each $T_e : |\mathbb{C}| \to |\mathbb{C}|$ to a functor $T_e : (\mathbb{C} \to \mathbb{C})$ by defining the action on morphisms $T_e(f) = (\eta \circ f)_e^\dagger$; this makes η a natural transformation $\eta : \mathsf{Id} \to T_1$. See McDermott and Uustalu [14] for more on graded monads.

For each monoid E and category $\mathbb{C}$, E-graded monads on $\mathbb{C}$ themselves form a category $\mathsf{GrdMnd}(E, \mathbb{C})$, where a morphism $T \to U$ is an E-indexed family of natural transformations $\alpha_e : T_e \to U_e$ which commutes with the units and Kleisli extensions in the following sense:



For each grade $e \in E$, we have a forgetful functor $R_e : \mathsf{GrdMnd}(E, \mathbb{C}) \to [\mathbb{C}, \mathbb{C}]$ which sends T to T_e. For $\mathbb{N}$-graded monads in particular, this forgetful functor at grade 1 has a left adjoint which is closely related to F_*:

Theorem 3.5. *The forgetful functor $R_1 : \mathsf{GrdMnd}(\mathbb{N}, \mathbb{C}) \to [\mathbb{C}, \mathbb{C}]$ has a left adjoint, which given $F : \mathbb{C} \to \mathbb{C}$ constructs the graded monad $n \mapsto F^n$.*

Proof. This construction is functorial: given a morphism in $\alpha : F \to G$, we define an $\mathbb{N}$-graded monad morphism $\alpha^n : F^n \to G^n$ by recursion on n:

$$\alpha^0 = \mathsf{id}$$
$$\alpha^{n+1} = G(\alpha^n) \circ \alpha_{F^n}$$

or equivalently $\alpha^{n+1} = \alpha_{G^n} \circ F\alpha^n$, by the naturality of α. Note in particular that $\alpha^1 = \alpha$. We construct a natural bijection on Homsets $\mathsf{Hom}_{\mathsf{GrdMnd}(\mathbb{N},\mathbb{C})}(F^-, M) \cong \mathsf{Hom}_{[\mathbb{C},\mathbb{C}]}(F, R_1(M))$: given $\alpha : F^- \to M$, we have $\alpha_1 : F \to R_1(M)$, and given $\beta : F \to R_1(M)$, we construct $\widehat{\beta}_n : F^n \to M_n$ by recursion on n:

$$\widehat{\beta}_0 = \eta^M : \mathsf{Id} \to M_0$$
$$\widehat{\beta}_{n+1} = (\widehat{\beta}_n)_1^\dagger \circ \beta_{F^n} : F \circ F^n \to M_n$$

Using the fact that α is a graded monad morphism, and $(\eta_X)^\dagger_e = \mathsf{id}_{T_e X}$, these constructions are seen to be inverse to one another. □

The concrete construction of F_* can be slightly generalised to arbitrary $\mathbb{N}$-graded monads over F at some grade k: Given any $\mathbb{N}$-graded monad G together with a natural transformation $\alpha : F \to G_k$, the coproduct $\sum_{n\in\mathbb{N}} G_n$ can be turned into an $(\mathsf{Id} + \_ \circ F)$-algebra induced by $\sigma : X \to \sum_{n\in\mathbb{N}} G_n(X)$ and $\rho_n : \sum_{n\in\mathbb{N}} G_n(F(X)) \to \sum_{n\in\mathbb{N}} G_n(X)$, where σ is defined to be the the composite

$$X \xrightarrow{\eta_X} G_0(X) \xrightarrow{\mathsf{in}_0} \sum_{n\in\mathbb{N}} G_n(X)$$

and ρ_n the composite

$$G_n(F(X)) \xrightarrow{G_n(\alpha_X)} G_n(G_k(X)) \xrightarrow{\mu_{n,k}} G_{n+k}(X) \xrightarrow{\mathsf{in}_{n+k}} \sum_{n\in\mathbb{N}} G_n$$

where the multiplication $\mu_{n,k} : G_n(G_k(X)) \to G_{n+k}(X)$ can be defined for arbitrary graded monads as $\mu_{n,k} = (\mathsf{id}_{G_n X})^\dagger_k$.

4 A Container-Theoretic Formalisation of Snoc-Trees

We have seen in Section 3.1 that snoc-trees can be understood as special cases of ordinary trees. The converse is intuitively not the case: all growth points in a snoc-tree exist at the same depth, whereas this is not the case in an ordinary tree. Thus, in general, we cannot turn normal trees into snoc-trees. However, we would like to prove that for *finitary* trees, we can indeed turn normal trees into snoc-trees. We call this process snocification. In order to state what we mean by finitary trees, we turn to the theory of containers [1], also known as polynomial functors [9, 16].[2] In this section, we restrict attention to endofunctors on the category of sets.

Definition 4.1. *A* container (S,P) *is given by a set* S *and an* S*-indexed family of sets* $P : S \to \mathsf{Set}$*. The* extension *of the container* (S,P) *is the functor* $[\![S,P]\!] : \mathsf{Set} \to \mathsf{Set}$ *given by* $[\![S,P]\!](X) = \sum_{s\in S} (P(s) \to X)$*.*

For a container (S,P), we think of S as the set of possible *shapes* of the container, and $P(s)$ as the set of *positions* to fill with data for a given position $s \in S$. This is made precise by the extension functor $[\![S,P]\!]$, which indeed says that an element in the container $[\![S,P]\!](X)$ is given by first choosing a shape s, and then filling all the positions $P(s)$ with data from X.

Lemma 4.2 (Abbott et al. [1])**.** *The identity functor is represented by the container* $(\mathbf{1},\mathbf{1})$*. Given containers* (S,P) *and* (T,Q)*, the product of their extensions is represented by the container* $(S \times T, (s,t) \mapsto P(s) + Q(t))$ *while their coproduct is represented by the container* $(S+T,[P,Q])$*. The composition of the extensions of* (S,P) *and* (T,Q) *is represented by the container* $([\![S,P]\!]T, (s,h) \mapsto \sum_{p\in P(s)} Q(h(p)))$*.*

The above lemma gives us the ingredients we need to consider the rank 2 definitions of F^* and F_* restricted to container functors: we would hope that if F is a container, then so is F^* and F_*.

[2] We acknowledge categorical notions of size such as accessibility and finite presentability, but we also want to work with containers for their concreteness.

Theorem 4.3. *Let (S,P) be a container.*

(i) The extension of the container (S^*,P^*), where S^* the least solution of

$$S^* \cong \mathbf{1} + \sum_{s:S} (Ps \to S^*)$$

and

$$\begin{aligned} P^*(\mathsf{inl} *) &\cong \mathbf{1} \\ P^*(\mathsf{inr}(s,f)) &\cong \sum_{p:P(s)} P^*(fp), \end{aligned}$$

is the initial algebra of $\mathsf{Id} + [\![S,P]\!] \circ \_$.

(ii) The extension of the container (S_*,P_*), where S_* and P_* form the least family satisfiying the equations below, is the initial algebra of $\mathsf{Id} + \_ \circ [\![S,P]\!]$.

$$\begin{aligned} S_* &\cong \mathbf{1} + \sum_{s_*:S_*} (P_*(s_*) \to S) \\ P_*(\mathsf{inl} *) &\cong \mathbf{1} \\ P_*(\mathsf{inr}(s_*,f)) &\cong \sum_{p_*:P_*(s_*)} P(f(p)) \end{aligned}$$

Proof. The result (i) is proven in Gambino and Kock [9]. For (ii), this follows from Lemma 4.2. □

The equations of S^* is a simple inductive definition, with P^* a (large) recursive definition. The equations for S_* and P_* can be understood *inductive-recursively* [7]: S_* is defined inductively, simultaneously with the recursive definition of P_*. Inductive-recursive definitions are proof-theoretically strong, but a more frugal presentation of S_* and P_* is also possible by grading the definition over $\mathbb{N}$. We define $S'_* : \mathbb{N} \to \mathsf{Set}$ and $P'_* : S_*(n) \to \mathsf{Set}$ as follows:

$$\begin{aligned} S'_*(0) &= \mathbf{1} \\ S'_*(n+1) &= \sum_{s_*:S'_*(n)} (P'_*(s_*) \to S) \\ P'_*(x : S'_*(0)) &= \mathbf{1} \\ P'_*((s_*,f) : S'_*(n+1)) &= \sum_{p_*:P'_*(s_*)} P(f(p_*)) \end{aligned}$$

We can then show that $S_* \cong \sum_{n\in\mathbb{N}} S'_*(n)$ and $P_*(s_*) = P_*(s_*)$, modulo this isomorphism. This gives a foundationally less strong definition of S_* and P_* which is easy to work with as well.

The elements of S_* represents snoc-trees, with growth points given by P_* — more precisely, elements of $S'_*(n)$ represents snoc-trees of height n. If we substitute a tree for each growth point of an ordinary tree, we again get an ordinary tree; this corresponds to the multiplication of a monad structure on trees. However the same is not true for snoc-trees: the resulting tree might not be a snoc-tree any more, because

different subtrees might have different height. However, if the snoc-tree has no growth points, we can pad it to a tree of higher height with a simple function

$$\mathsf{pad} : \{s \in S'_*(n) \mid P'_*(s) = \emptyset\} \to S'_*(n+1)$$

defined by $\mathsf{pad}(s) = (s, !)$ where $! : \emptyset \to S$ is the unique function out of the empty set.

If we now have a finite number of snoc-trees, then we can always use this padding operation to make sure that they all have the same height. In particular, if we start with an ordinary tree with finite branching degree, then we can translate it to a snoc-tree: using the elimination principle for ordinary trees, we can first translate all the subtrees, then pad them out to the same height k, and put them together as a snoc-tree. This proves the following theorem:

Theorem 4.4. *Given $t \in [\![S,P]\!]^*(X)$ where $P(s)$ is finite for each s, there is a corresponding snoc-tree $t' \in S_*$ with the same labels as t. Together with Lemma 3.2, this gives a retraction of trees onto snoc-trees: if we translate a snoc-tree to an ordinary tree and then back to a snoc-tree again, we end up with the same tree.*

5 Sampling

Sampling is about generating data according to a given distribution. Recall from the introduction that we will assume that D is a distribution monad, and let R be a monad for randomness. We work in Agda to ensure our definitions are well typed and, to avoid commitment to specific implementations, we represent D and R as postulates

```
postulate
  R : Set → Set
  _>>=_ : A B : Set → R A → (A → R B) → R B
  return : A : Set → A → R A

  D : Set → Set
  mapD : A B : Set → (A → B) → D A → D B
  reindexD : A B : Set → (A → B) → D B → D A
```

Slightly surprisingly, our development so far does not require D to actually be a monad, but is does require a contravariant and well as covariant action. Standard definitions of distributions support these actions. As mentioned earlier, sampling for a type A means providing a function $\mathsf{D}A \to \mathsf{R}A$ which takes as input a distribution over A and returns an effectful element of A. We assume we can sample on Bool by making another assumption

```
postulate
  sampleBool : D Bool → R Bool
```

A possible implementation is to ask for a more basic primitive `flip : R Bool` for generating a random bit, and to then use rejection sampling: use `flip` a finite but unbounded number of times to generate the

digits of a real number between 0 and 1. If the probability of ‘false‘ in the given distribution is greater than this number, return false, otherwise true.

Next we do a simple example of compositional sampling by lifting sampling from a type A to the type Maybe A:

```
maybeShape : A : Set → Maybe A → Bool
maybeShape nothing = true
maybeShape (just x) = false

sampleMaybe : A : Set → (D A → R A) → D (Maybe A) → R (Maybe A)
sampleMaybe sampleA ϕ = do
  b ← sampleBool (mapD maybeShape ϕ)
  if b
   then return nothing
   else do
    a ← sampleA (reindexD just ϕ)
    return (just a)
```

The function sampleMaybe takes the distribution $\phi : \mathsf{D}(\mathsf{Maybe}(A))$ and uses the function maybeShape and the functoriality of D to compute a probability distribution of whether the output will be a value or the constructor nothing. Sampling at booleans then extracts a boolean according to this distribution. If the result is true then nothing is the right value, otherwise we need to decide which value to output. This is done by sampling at type A and this requires ϕ to be restricted from a distribution over Maybe A to a distribution over A via the contravariant reindexD.

5.1 Sampling From Coproducts

Sampling from a coproduct type is similar to sampling from Maybe$A$. To sample from $A+B$ according to a distribution ϕ over $A+B$, we first compute how much of the probability mass of ϕ is over A and how much of the probability mass of ϕ is over B. This gives a probability distribution over Booleans, which we can sample from to know which summand we are going to produce a value from. Once that is known, we restrict ϕ to the probability distribution over that summand and then sample from that summand. Here is the implementation:

```
samplePlus : {A B : Set} → (D A → R A) → (D B → R B) → D (A ⊎ B) → R (A ⊎ B)
samplePlus sampleA sampleB ϕ = do
  s ← sampleBool (mapD [ (λ _ → true) , (λ _ → false) ] ϕ)
  if s
   then (do
    a ← sampleA (reindexD inl ϕ)
    return (inl a))
   else (do
    b ← sampleB (reindexD inr ϕ)
    return (inl b))
```

5.2 Sampling From Products

When it comes to sampling from products, we fundamentally use Bayes' Theorem, which allows us to sample a product by sampling for one value and then sampling for the other value given the result of the first sampling. Thus we have:

```
sampleTimes : {A B : Set} → (D A → R A) → (D B → R B) → D (A × B) → R (A × B)
sampleTimes sampleA sampleB ϕ = do
  a ← sampleA (mapD fst ϕ)
  b ← sampleB (reindexD (λ b → (a , b)) ϕ)
  return (a , b)
```

This naturally extends to dependent pair types, with the same implementation:

```
 sampleSigma : {A : Set}{B : A → Set}
             → (D A → R A)
             → (a : A → D (B a) → R (B a))
             → D (Σ A B) → R (Σ A B)
sampleSigma sampleA sampleB ϕ = do
  a ← sampleA (mapD fst ϕ)
  b ← sampleB (reindexD (λ b → (a , b)) ϕ)
  return (a , b)
```

By repeatedly applying `sampleTimes`, we can sample from finite products, but note that sampling from *infinite* products in general is difficult.

5.3 Sampling From Inductive Types

We finish by showing how to sample from inductive types. As a precursor we tackle the simpler case of lists to get a feel for the problem. Here is the code from the introduction again:

```
sampleList : {A : Set} → (D A → R A) → List A → D (List A) → R (List A)
sampleList sampleA xs ρ = do
  r ← sampleMaybe sampleA (mapD head ρ)
  case r of λ where
    nothing → return xs
    (just a) → sampleList sampleA (xs ++ [ a ]) (reindexD (a ∷_) ρ)
```

Hopefully some of this is code is now obvious. We take the distribution ϕ over $\mathsf{List}(A)$, and use it to create a distribution over $\mathsf{Maybe}A$, from which we sample to obtain either the empty list, or the first element a of the list to be produced. If the latter, we reindex ϕ to create a distribution from which to sample the rest of the list. The one new idea is that because lists are recursive, we need to store the list we have already constructed, attach the next element sampled, and then pass this new list on to the next recursive call until termination is reached.

Generalising from lists to trees, we have another complication in that trees can have more than one recursive substructure. To sample from trees with labels from S and branching degree $P : S \to \mathsf{Set}$, we assume that we can decide if the growth points $P_*(x)$ of the snoc-tree built from (S,P) are empty or not, and that we can sample from $P_*(x)$-indexed products of S. Both of these assumptions are satisfied if $P(s)$ is finite for each $s \in S$, since then $P_*(x)$ will be finite as well. To sample from such a tree, we use an accumulator snoc-tree s_*. If $P_*(s_*)$ is empty, then we convert s_* back to an ordinary tree, and take this as our result. Otherwise we sample one more layer of snoc-tree using the sampler for products of S, and add this layer to our accumulator in a recursive call, conditioning the probability distribution accordingly. In code, it looks as follows:

```
sampleTree : ((x : S*) → P* x ⊎ (P* x → ⊥))
           → ({x : S*} → D (P* x → S) → R (P* x → S))
           → (s* : S*)
           → D (P* s* → Tree (S , P)) → R (Tree (S , P))
sampleTree P*-decide-Empty sampleS s* ϕ with P*-decide-Empty s*
... | inr P*-empty = return (toTree S P (s* , P*-empty))
... | inl _ = do
 g ← sampleS (mapD (root ∘_) ϕ)
 sampleTree P*-decide-Empty sampleS (cont s* g)
            (reindexD (λ f p* → sup (g p* , λ x → f (p* , x))) ϕ)
```

6 Conclusions and Future Work

We have presented the notion of snoc-trees, and developed their meta-theory using both initial algebra semantics and containers. We have shown how they can be used to give compositional sampling algorithms (similar to a variant of Boltzmann sampling [6]) where we build a universe of types, and show that if we can sample over Booleans, we can sample over all types in that universe.

There are a number of avenues of further research to consider. From a data type perspective, one can always extend the results in this paper to create snoc-variants of inductive families, inductive-inductive definitions, etc, or develop forms of short cut fusion for snoc-trees. All this seems straightforward. What most interests us, though, is the use of snoc-trees. We plan to complete our type-driven approach to sampling by formally verifying that the distribution of data created by our algorithms matches that of the input distribution. In doing this, we definitely want to extend our universe to contain final coalgebras — we see no difficulty and only time prevented us from including them in this paper. Further, we can consider the algebraic properties of sampling, e.g. to what extent is the collection of sampling functions natural? More generally, snoc-trees is a natural data structure when working with inductive data types that have substructures that are not independent, and thus must be grown in lock-step.

References

[1] Michael Abbot, Thorsten Altenkirch & Neil Ghani (2005): *Containers: Constructing strictly positive types. Theoretical Computer Science 342 (1), 3-27.*

[2] Andreas Abel, Ralph Matthes & Tarmo Uustalu (2005): *Iteration and coiteration schemes for higher-order and nested datatypes*. Theoretical Computer Science 333.1-2: 3-66.

[3] Jiří Adámek, Stefan Milius & Lawrence S. Moss (2025): *Initial Algebras and Terminal Coalgebras: The Theory of Fixed Points of Functors*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

[4] Agda developers (2026): *Agda*. `https://agda.readthedocs.io/`.

[5] Luc Devroye (1986): *Non-Uniform Random Variate Generation*. Springer, doi:10.1007/978-1-4613-8643-8.

[6] Philippe Duchon, Philippe Flajolet, Guy Louchard & Gilles Schaeffer (2004): *Boltzmann Samplers for the Random Generation of Combinatorial Structures*. Combinatorics, Probability and Computing 13(4-–5), pp. 577—625, doi:10.1017/S0963548304006315.

[7] Peter Dybjer & Anton Setzer (1999): *A Finite Axiomatization of Inductive-Recursive Definitions*. In Jean-Yves Girard, editor: Typed Lambda Calculi and Applications, Springer, pp. 129–146, doi:10.1007/3-540-48959-2_11.

[8] Peter Dybjer & Anton Setzer (2003): *Induction-recursion and initial algebras*. Annals of Pure and Applied Logic 124(1-3), pp. 1–47.

[9] Nicola Gambino & Joachim Kock (2013): *Polynomial functors and polynomial monads*. Mathematical Proceedings of the Cambridge Philosophical Society 154(1), pp. 153—192, doi:10.1017/S0305004112000394.

[10] Neil Ghani, Peter Hancock & Dirk Pattinson (2009): *Continuous Functions on Final Coalgebras*. Electronic Notes in Theoretical Computer Science 249, pp. 3–18.

[11] Joseph Goguen (1974): *Initial Algebraic Semantics. IEEE Conf. Rec. SWAT*.

[12] Bart Jacobs (2018): *From probability monads to commutative effectuses*. Journal of Logical and Algebraic Methods in Programming 94, pp. 200–237, doi:10.1016/j.jlamp.2016.11.006.

[13] Patricia Johann & Neil Ghani (2009): *A principled approach to programming with nested types in Haskell*. Journal of Higher Order Symbolic Computation 22(2), pp. 155–189, doi:10.1007/S10990-009-9047-7.

[14] Dylan McDermott & Tarmo Uustalu (2022): *Flexibly graded monads and graded algebras*. In: International Conference on Mathematics of Program Construction, Springer, pp. 102–128.

[15] Eugenio Moggi (1991): *Notions of computation and monads*. Information and Computation 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[16] Nelson Niu & David I. Spivak (2025): *Polynomial Functors: A Mathematical Theory of Interaction*. London Mathematical Society Lecture Note Series, Cambridge University Press.

[17] Jan Rutten (2000): *Universal coalgebra: a theory of systems*. Theoretical Computer Science 249(1), pp. 3–80, doi:10.1016/S0304-3975(00)00056-6.