Implementing the Opetopes:
Programming with Higher Dimensional Trees

Eric Finster

May 22, 2018

Abstract

The opetopes, introduced by Baez and Dolan in [BD97] as a part of their proposed definition of higher dimensional category are closely related to to notion of higher dimensional tree of [Rog03] and [GK07]. In these notes, we try to spell out that relationship more explicitly, deriving in the process a compact representation for (labelled) opetopes in terms of simple inductive types. We explain various associated algorithms for manipulating the resulting structure.

Contents

1 Introduction

The *opetopes*, a family of polytopes introduced by Baez and Dolan in [BD97] as a part of their proposed definition of higher dimensional category, have deep links to the theory of inductive datatypes of the kind manipulated in modern functional programming lanugages. This relationship is perhaps most clearly spelled out in [KJBM07], which provides a definition of the opetopes in the language of *polynomial functors*, known sometimes as *indexed containers* in the computer science literature. It is more or less explicit in the definition that the opetopes are to be thought of as certain kinds of higher dimensional tree like structures. Viewed as such, however, their definition contains some subtleties.

There is, on the other hand, a more relaxed definition of what should constitue a higher dimensional tree. It appears already in the literature on linguistics and formal language theory (see, for example, [Rog03]). This definition was translated a datatype derivation in [GK07]. Roughly speak, the type of n-dimensional trees labeled by a type A can be seen as a natural continutation of the sequence of datatypes

$$A, \mathsf{List}A, \mathsf{Tree}A, \ldots$$

Our goal in these notes is to examine the relationship between these two approaches and to exploit the connections to derive useful representation of opetopes which can be implemented in modern functional languages.

2 Higher Dimensional Trees

2.1 Geometric Motivation

Our first task will be to derive a definition of higher dimensional tree using nothing more that some basic geometric intuition. More specifically, we would like to define, for each type A and each natural number n, a type $\mathsf{Tr}\,A\,n : \mathsf{Set}$ whose inhabitants are the n-dimensional trees with nodes labelled by elements of A. Said otherwise, we would like to fill in the definition of the type family

```
data Tr (A : Set) : ℕ → Set
```

Our strategy will be to work up to the correct definition by looking at some low-dimensional cases until we find a common abstraction.

With this in mind, let us begin by considering 0-dimensional trees. Geometrically speaking, we expect the only 0-dimensional tree to be a point. Hence, allowing our point to be labelled with an element of some arbirary type A, it seems clear that the type of 0-dimensional trees labelled by A is simply A itself. On the other hand, in order to have a genuine type constructor in each dimension so as to make generalization easier, let us adopt the following definition

```
data Tree₀ (A : Set) : Set where
  pt : A → Tree₀ A
```

The reader will notice that the type $\mathsf{Tree}_0\,A$ is indeed isomorphic to A itself, in line with our considerations above. Moreover, a reasonable geometric representation of an element of this type, say the element $\mathsf{pt}\,4 : \mathsf{Tree}_0\,\mathbb{N}$, would simply be a labelled point:

With the 0-dimensional case well in hand, let us consider 1-dimensional trees. Here too, the geometric intuition seems clear: a tree ought to be considered 1-dimensional exactly when it is *linear*, that is, when each node of the tree has at most one descendant. Moreover, if we label the nodes of a linear tree with elements of A, we should expect have a type isomorphic to just the type of *lists* of elements of A. Let us take, therefore, the following definition:

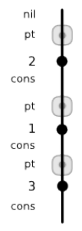
```
data Tree₁ (A : Set) : Set where
  nil : Tree₁ A
  cons : A → Tree₀ (Tree₁ A) → Tree₁ A
```

Notice that this definition is identical to the standard definition of the type of lists except for the inclusion of the Tree_0 constructor in the second argument of cons. As we have seen that Tree_0 is isomorphic to the identity functor, this makes no difference and our type is indeed isomorphic to $\mathsf{List}\,A$.

Let us again attempt to render a term of this type geometrically. For example, we might depict the term

```
ex₁ : Tree₁ ℕ
ex₁ = cons 3 (pt (cons 1 (pt (cons 2 (pt nil)))))
```

as follows:



In this picture, I have added the grey boxes to indicate that, after the appearence of a node constructor with a piece of data, the *descendants* of the linear tree (of which, of course, there is exactly one) are organized by the previous Point type constructor.

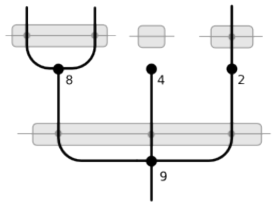
Next, we arrive at dimension 2, where we find planar trees. The appropriate data type definition is the following:

```
data Tree (A : Set) : Set where
  leaf : Tree A
  node : A → Tree₁ (Tree A) → Tree A
```

An example of a 2-tree would be something like:

```
ex₂ : Tree ℕ
ex₂ = node 9 (cons (node 8 (cons leaf (pt (cons leaf (pt nil)))))
  (pt (cons (node 4 nil)
  (pt (cons (node 2 (cons leaf (pt nil))) (pt nil))))))
```

with representation

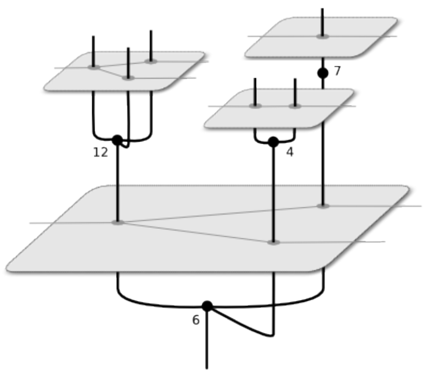


Noticing the pattern, we make the following general definition:

```
data Tr (A : Set) where
  obj : A → Tr A 0
  lf : {n : ℕ} → Tr A (suc n)
  nd : {n : ℕ} → A → Tr (Tr A (suc n)) n → Tr A (suc n)
```

In other words, a n-tree is either a leaf, or a piece of data together with an $(n-1)$-tree of descendants. Said still another way, the descendants of a node in an n-tree are equipped with the structure of an $(n-1)$-tree. It is this structure on the descendants which the "grey boxes" in the previous diagrams have been illustrating.

For a more striking example, here is a 3-tree.



2.2 Stabilization

Our indexed inductive type of higher dimensional trees corresponds quite well to the intuition and functions quite well in the dependently typed setting. However, for implementation in a more conventional functional language like Ocaml, Haskell or Scala, keeping track of the dimension as part of the structure turns out to be quite inconvenient.

A nice way around this problem is to imagine taking the limit $n \to \infty$. Doing so results in the following type, which I will refer to as the type of *stable trees*.

```
data STree (A : Set) : Set where
  lf : STree A
  nd : A → STree (STree A) → STree A
```

As the parameter n was indexing the dimension of the tree, a sometimes useful geometric intuition is provided by thinking of this as the type of trees embedded in ∞-dimensional space. In fact, this definition already appears in [BM98] as the type of "Bushes", though the corresponding geometric interpretation is absent.

Notice that the "point" constructor in dimension 0 now becomes redundant, as it can be encoded using the following:

```
spt : {A : Set} → A → STree A
spt a = nd a lf
```

2.3 Maps and Traversals

It should be quite clear that the data structures thus defined are in fact functors. For example, in the stable case we have:

```
stree-map : {A B : Set} (f : A → B) → STree A → STree B
stree-map f lf = lf
stree-map f (nd a sh) = nd (f a) (stree-map (stree-map f) sh)
```

This is, of course, a useful fact about higher dimensional trees, but for our purposes below, we will need something slightly more sophisticated.

As we will see below, we are interested in certain monad structures on the type of higher dimensional trees. However, the various monad multiplications will be defined recursively in terms of multiplications in previous dimensions. As a result, without a complicated dependently typed setup, we will not be able to expresses the actual type of our monadic operations.

A way around this problem is to allow our multiplication to be *partial*. This, however, has the following consequence: we are often confronted with the problem of applying some kind of computation at every node of a higher dimensional tree, with the possibility that the computation might fail. If we model this computation as a function like $f : A \to \mathsf{Option} B$, the mapping this over a given $t : \mathsf{STree} A$ will produce an element

$$\mathsf{stree\text{-}map} f t : \mathsf{STree}(\mathsf{Option} B)$$

That is, we obtain the stable tree *decorated* with the computations we wanted to do. But how to we actually *run* them and produce an element of type $\mathsf{Option}(\mathsf{STree} B)$ which fails if any one of the computations at each node fails and succeeds with the tree of results if each computation succeeds?

In short we need a notion of generic traversal as developed for example in [GO09].

2.4 Addresses and Zippers

And important technique in functional programming consists of editing an immutable data structure using the notion of a *zipper*. In manipulating higher dimensional trees, having the same tools available will make life considerably easier. Hence we pause here to derive these types and we will put them to use in the next section.

First, let us start thinking about what data we need to determine a node in an n-dimensional tree. We will call such a thing an n-dimensional *address*. First of all, a 0-dimensional tree, that is, a point, has by definition a unique node. So the type of 0-addresses is just a point. It is not hard to see that an address in a 1-tree corresponds to a natural number which we can think of as the index of the desired node. Moreover, an address into a 2-tree is a *list* of natural numbers with each successive element of the list telling us which branch to follow.

If we identify the natural numbers with lists of unit type, then the pattern becomes clear:

```
Addr : ℕ → Set
Addr 0 = ⊤
Addr (suc n) = List (Addr n)
```

This is to say that, in order to determine a node in an n-tree, we must have a list of address into $(n-1)$-trees, with each successive address telling us which branch to follow.

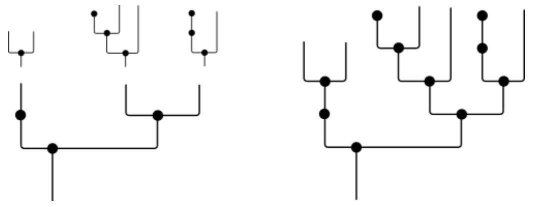
Passing to the stable case, we find

```
data SAddr : Set where
  dir : List SAddr → SAddr
```

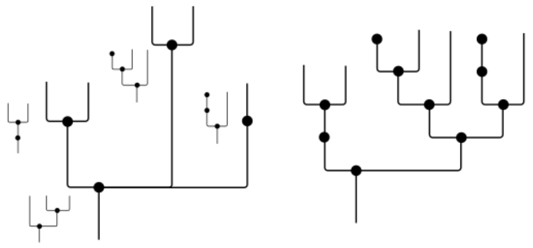
2.5 Grafting and Joining

There are two fundamental operations on the collection of stable trees that we will need to understand in order to arrive at a definition of opetope.

This first looks like this:



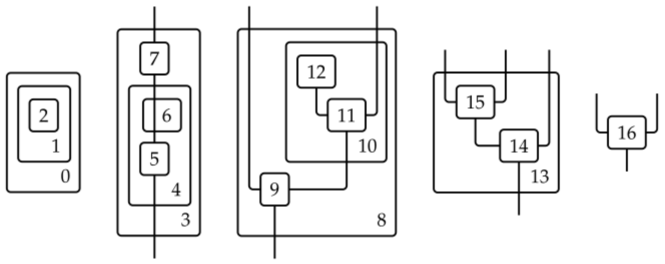
The second looks like this:



3 Opetopes

3.1 Nestings and Complexes

Our goal is to work towards are representation of opetopes like the following, and introduced in [KJBM07].



It is convenient to introduce the following variation of stable tree which allows us to store a piece of data a a leaf:

```
data Nesting (A : Set) : Set where
  dot : A → Nesting A
  box : A → STree (Nesting A) → Nesting A
```

Finally we are ready to give the definition which will encode our opetopes. Since not every element of this type is in fact an opetope, I will call it a "complex".

```
Complex : (A : Set) → Set
Complex A = List (Nesting A)
```

The list of nestings corresponding to the above complex is the following:

```
c₀ = box 0 (nd (box 1 (nd (dot 2) lf)) lf)

c₁ = box 3 (nd (box 4 (nd (dot 5) (nd (nd (box 6 lf) (nd lf lf)) lf)))
  (nd (nd (dot 7) (nd lf lf)) lf))

c₂ = box 8 (nd (dot 9) (nd (nd (box 10 (nd (dot 11)
  (nd lf (nd (nd (nd (dot 12) lf) (nd lf lf)) lf))))
    (nd lf (nd lf lf))) (nd (nd lf (nd lf lf)) lf)))

c₃ = box 13 (nd (dot 14) (nd lf (nd (nd (nd (dot 15)
  (nd lf (nd lf (nd (nd (nd lf lf) (nd lf lf)) lf))))
    (nd lf (nd lf lf))) (nd (nd lf (nd lf lf)) lf))))

c₄ = dot 16
```

Indeed, the graphical representation can be viewed as merely a shorthand notation for the above rather unwieldy syntactical representation.

3.2 A Correctness Criterion

In this section, we finally complete our definition of opetopes in terms of the type of complexes introduced above.

3.3 Computing Faces

We now show how to compute faces of opetopes.

References

[BD97] John C Baez and James Dolan. Higher-dimensional algebra iii: n-categories and the algebra of opetopes. *arXiv preprint q-alg/9702014*, 1997.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction*, pages 52–67. Springer, 1998.

[GK07] Neil Ghani and Alexander Kurz. Higher dimensional trees, algebraically. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science*, pages 226–241, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[GO09] Jeremy Gibbons and Bruno C d S Oliveira. The essence of the iterator pattern. *Journal of functional programming*, 19(3-4):377–402, 2009.

[KJBM07] Joachim Kock, André Joyal, Michael Batanin, and Jean-François Mascari. Polynomial functors and opetopes. *arXiv preprint arXiv:0706.1033*, 2007.

[Rog03] James Rogers. Syntactic structures as multi-dimensional trees. *Research on Language and Computation*, 1(3-4):265–305, 2003.