# Opetopes and Higher Dimensional Trees

Eric Finster

May 18, 2018

We describe the relationship between opetopes and higher dimensional trees, exploiting the latter to derive a useful representation of opetopes in terms of inductive types.

## 1 Higher Dimesional Trees

We begin with an intuitive description of the type of higher dimensional trees. As we are interested in *labelled* higher dimensional trees, we are going to try to describe, for each $n$, an inductive type whose labels come from a fixed type $A$. For examples, we will often take $A = \mathbb{N}$ for concreteness.

### 1.1 Geometric Motivation

In order to work our way up to the definition for each $n$, let us examine some of the lowest dimensional cases. For example, what should be a 0-tree decorated with elements of $A$? Well, unsurprisingly, a 0-tree is just a point. So the type of 0-trees labelled with a point of the type $A$ is simply $A$ itself. Put another way, the functor which, given a type $A$ returns the type of 0-trees in $A$ is simply the identity.

For our purposes, however, it will be useful to mark the point with a dummy constructor in order to better illustrate what is happening. This can be accomplished with the following definition:

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

We see from this definition that the only way to construct an element of Point $A$ is simply to provide an element of $A$. Here is an example

```
ex₀ : Point ℕ
ex₀ = pt 4
```

We can represent this piece of data graphically as follows:

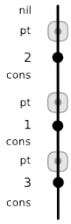
Now, let us pass to dimension 1. After a bit of reflection, it seems clear that a 1-tree should be *linear*. And hence if we allow decorations in our type $A$, we see that this type of 1-trees in $A$ is just the type of lists of elements of $A$. Here is the definition I propose:

```
data List (A : Set) : Set where
  nil : List A
  cons : A → Point (List A) → List A
```

An example element might be as follows:

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

and its corresponding geometric representation might be rendered as




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.

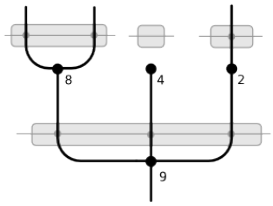
Next, we arrive at dimension 2, where we find planar trees. The appropriate data type definition is the following:

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

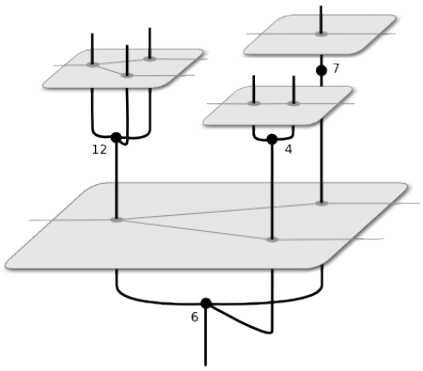



Noticing the pattern, we make the following general definition:

```
data Tr (A : Set) : ℕ → 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.




## 1.2 Stable Trees

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 $\infty$-dimensional space.

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

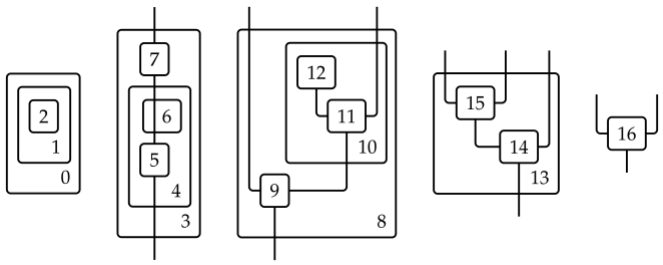
```
spt : {A : Set} → A → STree A
spt a = nd a lf
```

### 1.3 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.

## 2 Opetopes

Our goal is to work towards are representation of opetopes like the following:




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:

$c_0$ = box 0 (nd (box 1 (nd (dot 2) lf)) lf)

$c_1$ = 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))

$c_2$ = 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)))

$c_3$ = 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))))

$c_4$ = dot 16

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