A Formal Model of Hillview

June 2020

1 Basics

Notation	Meaning
M	Set of machines.
$\mathcal{E}$	Environment.
D	Type of data stored in a tree.
$Tree\langle D\rangle$	Tree with data of type D in leaves.
t	A tree instance.
(a, b)	The pair with values a and b.
l	A leaf node in a tree.
i	An internal node in a tree.
R	Monoid (set with associative operation $+_R$ and a zero 0_R); results produced by sketches.
$A[B]$	Partial functions (key-value maps) from B to A, where A and B are sets
$\mathbb{N}[A]$	A multiset over A.
T^*	Lists of values of type T.
[]	The empty list.
T	A "tuple"; usually $D = T^*$.
$\|t\|$	The interpretation of a tree as a multiset $\mathbb{N}[D]$.
$x \mapsto x + 1$	A function that maps x to $x + 1$.
$\{a \mapsto 2\}$	A multiset where the value a appears twice.

Table 1: Notations used in this document.

M is a countable set of machines. S are strings over some alphabet.

D denotes a type; the set of types includes (), $\mathbb{N}$, $\mathbb{B}$ — Booleans, strings, product types, sum types, etc. The type $A[B]$ is the type of finite maps from B to A.

We will use often the "object oriented" suffix notation for functions: $a.f(x)$ is the same as $f(a, x)$. A function with only 1 argument is also called a "property" and can be shown without parentheses in the suffix notation: $a.f$.

$\mathcal{E}$ is the "environment" that stores data on each machine. The environment provides a set of functions to read the data (think of reading the data from a file): $\mathcal{E}.\mathbf{read}_D : M \to S \to D$, where $d : D = \mathcal{E}.\mathbf{read}(m, '\mathbf{people}')$ is the data (of type D) residing on machine m in the file named $'\mathbf{people}'$.

We define two handy functions: $\mathbf{one}_R : R \to \mathbb{N}$ (for an arbitrary R), defined by $\mathbf{one}(x) = 1$, and the identity function $\mathbf{id}_R : R \to R$, $\mathbf{id}(x) = x$.

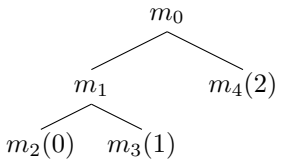
2 Computation trees

We deal with typed *computation trees*: $\mathit{Tree}\langle D\rangle$ is the set of trees with a value of type D in each leaf node (all leaves of a tree must have the same type of data).

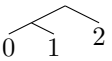
Nodes N_D in a tree of type D are either internal nodes I_D or leaves L_D, so $N_D = I_D \oplus L_D$. A tree is defined by its root $t \in N_D$. Every tree node has a **machine** property, which is the machine where the tree node resides: $\mathbf{machine} : N_D \to M$.

In addition, each leaf node has a **data** property: the data value it stores locally $\mathbf{data} : L_D \to D$.

The children of an internal node $n \in I_D$ are given by a finite partial function $\mathbf{children} : I_D \to N_D[M]$ that takes a machine and returns a child node that must reside on that machine (for any $i \in I_D, i.\mathbf{children}(m).\mathbf{machine} = m$). An internal node can have zero children. In the following figure we display a tree $t \in \mathit{Tree}\langle\mathbb{N}\rangle$; we show for each node its machine m_x; for leaf nodes we also display the data value, a number:



If we do not particularly care about the machine where a node resides; then we can display just the leaf values:



We will consider an *interpretation function* $|.|$ over trees that maps a tree to a multiset of values over D, the multiset of the values in all leaves. $|.| : \mathit{Tree}\langle D\rangle \to \mathbb{N}[D]$, defined recursively as follows:

For a leaf node $|l| = \{l.\mathbf{data} \mapsto 1\}$ (the **data** value with a count of 1). For an internal node: $|i| = \cup_{c \in i.\mathbf{children}} |c|$, where we use the multiset union.

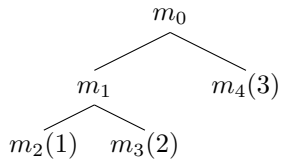
2.1 Transforming trees (map)

Given a function $f : D_0 \to D_1$, the **map** operator produces a function between trees: $\mathbf{map} : (D_0 \to D_1) \to \mathit{Tree}\langle D_0\rangle \to \mathit{Tree}\langle D_1\rangle$, defined as follows:

For a leaf node: $l \in L_{D_0}$, $l.\mathbf{map}(f).\mathbf{data} = f(l.\mathbf{data})$, and $l.\mathbf{map}(f).\mathbf{machine} = l.\mathbf{machine}$ (the result is a leaf on the same machine, with data produced by applying f to the existing data).

For an internal node $i \in I_{D_0}$ we have recursively $i.\mathbf{map}(f).\mathbf{children}(m) = i.\mathbf{children}(m).\mathbf{map}(f)$ and $i.\mathbf{map}(f).\mathbf{machine} = i.\mathbf{machine}$. (same machine, children produced recursively).

If t is the tree defined above, $t.\mathbf{map}(x \mapsto x + 1)$ produces t_1:



Note that each node resides on the same machine as the "source" node that it was produced from.

2.2 Growing trees (flatmap)

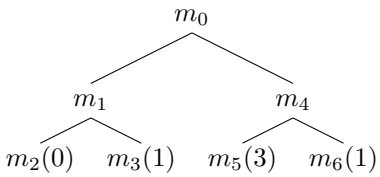
Let us consider a function that given a leaf node produces a tree : $f : L_{D_0} \to \mathit{Tree}\langle D_1 \rangle$. Note that this function can "use" the environment of the node to produce it's result (e.g., by reading the list from the environment); all the leaves in the result must have the same type D_1.

We can define the **flatmap** operator, which produces a function between trees: $\mathbf{flatmap} : (L_{D_0} \to \mathit{Tree}\langle D_1 \rangle) \to \mathit{Tree}\langle D_0 \rangle \to \mathit{Tree}\langle D_1 \rangle$, defined as follows:

For a leaf node $l \in L_{D_0}$ **flatmap** produces a new internal node $l.\mathbf{flatmap}(f).\mathbf{children}(m) = f(l)$, where in the tree produced by f each new node resides on the "right" machine.

For an internal node $i \in I_{D_0}$ we have recursively $i.\mathbf{flatmap}(f).\mathbf{children}(m) = i.\mathbf{children}(m).\mathbf{flatmap}(f)$.

As a example, consider a flatmap operator that takes as an argument function that transforms the leaf node on machine m_4 into the sub-tree $[m_5 \mapsto 3, m_6 \mapsto 1]$ and does not change the other leaves. Applied to t this produces:



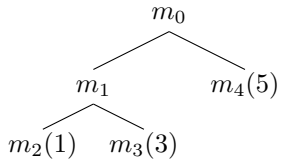
2.3 Combining multiple trees (zip)

Given two isomorphic trees $t_0 \in \mathit{Tree}\langle D_0 \rangle$ and $t_1 \in \mathit{Tree}\langle D_1 \rangle$ with the corresponding nodes on the same machines, and a function $c : D_0 \times D_1 \to D$, we can define the zip of the two trees as follows:

For leaf nodes $\mathbf{zip}(c)(l_0, l_1).\mathbf{data} = c(l_0.\mathbf{data}, l_1.\mathbf{data})$.

For internal nodes the zip is applied recursively to corresponding children: $\mathbf{zip}(c)(i_0, i_1) = [m \mapsto \mathbf{zip}(c)(d_0, d_1) | m \in M \wedge ((m \mapsto d_0) \in i_0) \wedge ((m \mapsto d_1) \in i_1)]$.

As an example, let us apply $\mathbf{zip}((x, y) \mapsto x + y)$ to the trees t and t_1 to obtain:



(I believe that zip can be generalized to work with non-isomorphic trees by treating missing nodes as a particular tree that has no leaves, and missing leaves as having a distinguished value representing an empty set with values in D.)

2.4 Shrinking trees (prune)

Given a function $e : D \to \mathbb{B}$, we define a **prune** operator on trees that removes leaves where e returns "false":

For a leaf node

$$l.\mathbf{prune}(e) = \begin{cases} l & \text{if } e(l.\mathbf{data}) \\ \phi & \text{otherwise} \end{cases}$$

.

For an internal node: $i.\mathbf{prune}(e).\mathbf{children} = \cup_{c \in i.\mathbf{children}} c.\mathbf{prune}(e)$.

3 Aggregating tree data

So far all the operations that we had were converting trees into trees. Aggregation is the only operation that can produce a "simple" value from a tree: a summary of the data stored in the tree's leaves.

3.1 Sketches

Consider a set of values D and a monoid $(R, +, 0)$ (most often we work with commutative monoids, but commutativity is not required). A *sketch* is an operator $\mathbf{sketch}(s_R)$ defined by a function $s_R : D \to R$. Note that the monoid R is part of the sketch definition; when this is clear from the context we write s, omitting the subscript R. But one can define multiple monoids over the same underlying set (e.g., for $\mathbb{N}$ the operation can be either addition or "max"), so it is very important to understand which monoid is being used to compute a specific sketch. Given a tree $t \in \mathit{Tree}\langle D \rangle$, we define a *sketch operator* that transforms trees into values in R as a function $\mathbf{sketch}(s) : \mathit{Tree}\langle D \rangle \to R$ defined recursively as follows:

For leaves $l.\mathbf{sketch}(s) = s(l.\mathbf{data})$.

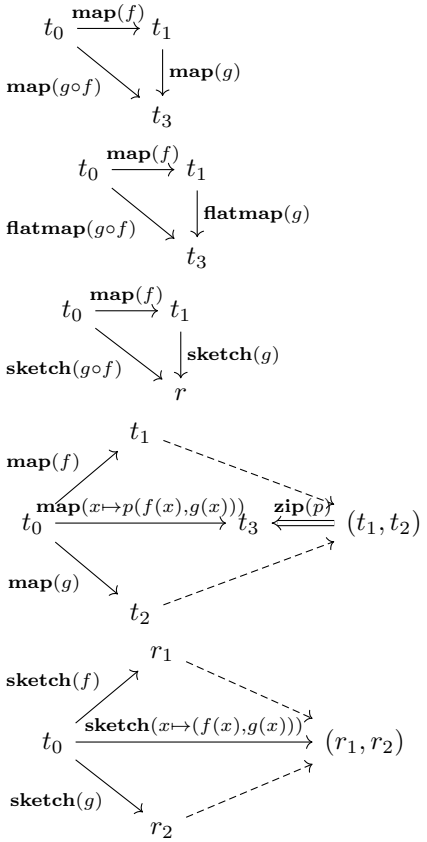
For internal nodes $i.\mathbf{sketch}(s) = \sum_{c \in i.\mathbf{children}} c.\mathbf{sketch}(s)$, where the sum is done using the monoid operation. Notice that the machine where the nodes reside has no effect on the sketch result.

As an example, consider the monoid $(\mathbb{N}, +, 0)$ of natural numbers with addition. Then we can apply the sketch operator to the above tree t with the identity function as an argument to obtain the sum of values in all leaves $t.\mathbf{sketch}(\mathbf{id}) = 0 + 1 + 2 = 3$.

If we consider sketches as being applied to tree interpretations we notice that all sketches are linear transformations between $\mathbb{N}[D]$ and R. If the monoid R is commutative then the result of applying a sketch to a tree does not depend on the ordering of children.

4 Properties of operators

The following commutative diagrams describe properties of these operators:

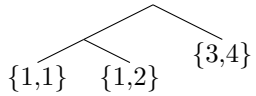


5 Additional structure on D

We will now put additional structure on D; in particular, we will consider types where D is a list of values over a base type T, i.e., $D = T^*$. In this case we can also extend the interpretation function of a tree to treat it as a multiset over T: $|.| : \mathit{Tree}\langle T^* \rangle \to \mathbb{N}[T]$. With this interpretation sketches are still linear transformations between tree interpretations and the result monoid.

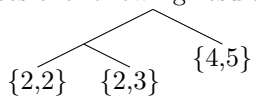
Given this structure we can define a few more interesting operations on trees in terms of the existing operations.

As an example, consider a tree $t_2 \in \mathit{Tree}\langle \mathbb{N}^* \rangle$, where each leaf contains a list of integers:



5.1 Maps

Recall that the **map** operators takes a function $m : D_0 \to D_1$; now let $D_0 = T_0^*$ and $D_1 = T_1^*$. Given a function $f : T_0 \to T_1$ we can perform the map of a map: $t_2.\mathbf{map}(\mathbf{map}(x \mapsto x + 1))$, where the inner map is the classic map operator defined over lists. This produces the following result:



5.2 Filters

Consider a predicate $p : T \to \mathbb{B}$. Define $\mathbf{filter}(p) : T \to T^*$ as

$$\mathbf{filter}(p)(e) = \begin{cases} \{e\} & \text{if } p(e) \\ \phi & \text{(empty set) otherwise} \end{cases}$$

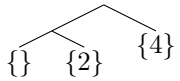
.

We have the classic list filtering operator: $\mathbf{filter} : (T \to \mathbb{B}) \to T^* \to T^*$:

$$\mathbf{filter}(p)(v) = \begin{cases} [] & \text{if } v = [] \\ \mathbf{cons}(\mathbf{head}(v), \mathbf{filter}(p)(\mathbf{tail}(v))) & \text{if } p(\mathbf{head}(v)) \,. \\ \mathbf{filter}(p)(\mathbf{tail}(v)) & \text{otherwise} \end{cases}$$

We can also use the tree **map** operators to perform filtering on tree leaves: $\mathbf{filter}(p)(t) = t.\mathbf{map}(\mathbf{filter}(p))$.

For example, given the predicate $p : \mathbb{N} \to \mathbb{B} = n \mapsto n \pmod 2 = 0$, the expression $t_2.\mathbf{map}(\mathbf{filter}(p))$ produces the following result:



5.3 Aggregates

We can compute interesting aggregate functions over the leaves of a tree using the tree **sketch** operator. For example, to compute the count of elements in a tree we can use the following "count" sketch: $R = (\mathbb{N}, +, 0)$, $\mathbf{count} : T^* \to \mathbb{N}$, $\mathbf{count}(x) = \sum_{e \in x} 1 = \sum_{e \in x} \mathbf{one}(e)$. We can count the number of values in t_2 as follows: $t_2.\mathbf{sketch}(\mathbf{count})$ to get the result 6 (there are 6 values in the leaves).

To compute the average value of the leaves we can use a more complex sketch function that computes a tuple composed of the sum and the count: $\mathbf{avg} = x \mapsto (\sum_{e \in x} \mathbf{id}(e), \mathbf{count}(x))$; by dividing the sum by the count we can

obtain the average. The sketch $\mathbf{sketch}(\mathbf{avg}) : \mathit{Tree}\langle\mathbb{N}\rangle \to \mathbb{N}\times\mathbb{N}$, where addition is applied pointwise.

5.4 Incremental sketches

Consider a function $\mathbf{inc} : T \to R$, where R is a monoid. We can define recursively the operator $\mathbf{fold}(\mathbf{inc}) : T^* \to R$ as

$$\mathbf{fold}(v) = \begin{cases} 0_R & \text{if } v = [] \\ f(\mathit{head}(v)) +_R \mathbf{fold}(\mathbf{inc})(\mathit{tail}(v)) & \text{otherwise} \end{cases}.$$

We call such functions "increments", because they define the contribution of an element $t \in T$ to the result R. Each increment function can be used to define an *incremental sketch* operator: $\mathbf{incsketch}_R(\mathbf{inc}) = \mathbf{sketch}_R(\mathbf{fold}(\mathbf{inc}))$. For example, the "count" sketch is defined as $\mathbf{incsketch}_{\mathbb{N}}(\mathbf{one}_{\mathbb{N}})$, and the "sum" sketch is $\mathbf{incsketch}_{\mathbb{N}}(\mathbf{id}_{\mathbb{N}})$.

5.5 Helper data structures

Sometimes, while we fold an increment over a set to produce a result in R we need to carry additional information, so we use a different monoid R', and some auxiliary functions to convert back and forth from R and R':

$(\mathbf{inc}_{R'}, \mathbf{done}_{R'})$, where

- $\mathbf{inc}_{R'} : T \to R'$ is the increment,
- $\mathbf{done}_{R'} : R' \to R$ is the function that extracts the final result.

The value $0_{R'}$ is used to generate the initial value for folding over R'. A new sketch operator $\mathbf{helpersketch}(f')$ can be defined, as follows $\mathbf{helpersketch}(f'_{R'}) : \mathit{Tree}\langle T^*\rangle \to R$, $\mathbf{helpersketch}(f') = \mathbf{sketch}_R(\mathbf{done}_{R'} \circ \mathbf{fold}_{R'}(\mathbf{inc}_{R'}))$.

An example for the use of **helperketch** is processing a list by sampling its elements with a given sampling rate. In this case a value of R' can hold the result produced so far (a value from R), but also the current state of the random number generator; the **inc** function skips some of the elements based on the sampling rate, and the **done** function scales the final counts by dividing with the sampling rate. (This scaling produces meaningful results only when the values produced by **inc** are bounded, e.g., when counting elements.)

5.6 Higher order incremental sketches

Let us consider the problem of computing the "histogram" of a dataset using a sketch. A histogram divides the data into a fixed, finite set of buckets n and counts the number of elements that fall into each bucket. The buckets can be defined abstractly by a function $\mathbf{bucket} : T \to [n]$, which assigns a value between 0 and $n-1$ to each element of T. A histogram sketch produces values in the monoid $\mathbb{N}^n$ (vectors of n integers, using pointwise element addition). We define an auxiliary (Kronecker delta) function $\delta_{\mathbb{N}}$, which receives an index, and

(an unused) value and produces a vector with a single 1 value, corresponding to the index: $\delta_{\mathbb{N}} : [n] \times T \to \mathbb{N}^n$: $\delta_{\mathbb{N}}(i,t) = v$, where $\begin{cases} v[j] = 0 & \forall j.i \neq j \\ v[i] = 1 \end{cases}$.

We can then define a histogram operator parameterized with the bucket function as the following sketch: $\mathbf{histogram}(\mathbf{bucket}) = \mathbf{incsketch}(\delta_{\mathbb{N}} \circ \mathbf{bucket})$.

We can further generalize this scheme: instead of producing a vector of numbers $\mathbb{N}^n$, we produce a vector of values over an arbitrary monoid R. Let us choose another function $f : T \to R$ and define a generalized version of δ with two parameters: $\delta_{f,R} : [n] \times T \to R^n$ function as: $\delta_{f,R}(i,t) = v$, where $\begin{cases} v[j] = 0_R & \forall j.i \neq j \\ v[i] = f(t) \end{cases}$.

We can now speak of a higher-order sketch $\mathbf{histogram} : \mathit{Tree}\langle T \rangle \to R^n$, parameterized with another sketch-defining function $f : T \to R$, defined by $\mathbf{histogram}(\mathbf{bucket}, f) = \mathbf{incsketch}(\delta_{f,R} \circ \mathit{bucket})$.

The standard histogram is just $\mathbf{incsketch}(\delta_{\mathbf{one},\mathbb{N}} \circ \mathbf{bucket})$. Let us denote $\eta : T \to \mathbb{N}^n = \delta_{\mathbf{one},\mathbb{N}} \circ \mathbf{bucket}$; then the standard histogram is $\mathbf{incsketch}(\eta)$.

5.7 Applications of higher-order sketches

5.7.1 Group by-sum

If we replace **one** with **id** in the above definition, we get a different kind of histogram, that sums up all values in a bucket, instead of counting them: $\mathbf{incsketch}(\delta_{\mathbf{id},\mathbb{N}} \circ \mathbf{bucket}) : \mathit{Tree}\langle \mathbb{N} \rangle \to \mathbb{N}^n$.

5.7.2 Two-dimensional and higher-dimensional histograms

We can even compute a two-dimensional histogram: given two bucket functions $\mathbf{bucket} : T \to [n]$, and $\mathbf{bucket}' : T \to [m]$, $\mathbf{incsketch}(\delta_{\eta,\mathbb{N}^n} \circ \mathbf{bucket}') = \mathbf{bucket}' : T \to [m]$, $\mathbf{incsketch}(\delta_{(\delta_{\mathbf{one},\mathbb{N}} \circ \mathbf{bucket}),\mathbb{N}^n} \circ \mathbf{bucket}') : \mathit{Tree}\langle T \rangle \to \mathbb{N}^{n \times m}$. This scheme is easily generalized for higher-dimensional histograms.

5.7.3 Random samples

Another interesting sketch uses reservoir sampling to extract random samples from a dataset. Reservoir sampling can be implemented as a sketch using a randomized function $\mathbf{reservoir} : T^* \to (T^n, \mathbb{N})$ that extracts $n^\dagger$ samples from a stream. The second value of the tuple returned by **reservoir** is the number of elements in the stream. Reservoir sampling initially fills up a vector of samples, and then proceeds to use a coin biased with a probability that keeps decreasing to decide whether to "replace" one of the saved samples. Merging two vectors $(v_0, c_0) \in (T^n, \mathbb{N})$ and $(v_1, c_1) \in (T^n, \mathbb{N})$ uses a biased coin with bias $c_0/(c_0 + c_1)$ to decide whether to keep an element from v_0 or v_1 (and sums up the counts).

[†]Or fewer, if the stream is short.

The higher-order bucketing sketch can be combined with the reservoir sketch to produce a vector of samples, one per bucket: $\mathbf{incsketch}(\delta_{(\mathbf{reservoir},(T^n,\mathbb{N}))} \circ \mathbf{bucket})$.

5.8 Other interesting sketches

In this section we describe briefly other interesting sketches that can be computed using this model.

5.8.1 First-k

This sketch is described by two parameters: a total order o over T, and a maximum size k. This sketch returns the $k^\ddagger$ largest elements according to the total order: $\mathbf{first}(o,k) : \mathit{Tree}\langle T^* \rangle \to T^k$. The monoid addition function is "merge sort".

5.8.2 Heavy hitters

This sketch is described by two parameters: an equality relation o over T, and a minimum frequency $0 < f \leq 1$, and it returns all tuples that compose at least a fraction $1/f$ of the multiset of leaf values. This can be implemented exactly with an algorithm such as Mishra-Gries, or approximately by sampling. Since Mishra-Gries can produce false positives, a second sketch can be used to compute the exact frequencies of the elements to eliminate the false positives.

5.8.3 Simple statistics

Given numeric data sketches can be used to compute simple statistics of the data: the min, max, count, average, statistical moments $(\sum_i e_i^k)/n$.

5.8.4 Approximate quantiles

Given a total order o over T and a parameter k this sketch can compute a set of approximate quantiles: k tuples that are approximately equi-distant in the sorted order produced by sorting all values in the leaves. This sketch is trivially implemented by executing the random sampling sketch from Section 5.7.3 selecting roughly k^2 samples, sorting the produced results and choosing the k empirical quantiles of this distribution (the sorting and re-sampling are done as a post-processing step on the sketch result). This can be used to compute the (approximate) median of a set, quartiles, deciles, etc.

5.8.5 Approximate quantiles over distinct values

This is the same problem as in the previous section, but in the sorted order we only need to consider each distinct element in T only once. This can be done

‡Or fewer if there are not enough elements.

using min-hashing, by applying a hash function to each value and keeping only the k values with the *smallest* hash values.

5.8.6 Hyper-log-log

The hyper-log-log sketch can be used to compute an estimate of the distinct elements in a multiset.

5.8.7 Centroids

The higher-order "histogram" sketch can be composed with the "average" to compute centroids of the values in each bucket; this is a step of the k-means algorithm.

5.8.8 Correlation matrix; Principal Component Analysis

Given a set of tuples T where each tuple is a vector of numbers $T = \mathbb{N}^k$, the correlation $k \times k$ matrix of all the "columns" can be computed using a sketch. This matrix can then be used for principal component analysis.