1. Trees

We use trees as the abstract view for our lenses; since the concrete data structure, strings, is ordered, and to support some properties of lenses that seem sensible intuitively, the trees differ from garden-variety trees in a number of ways:

- a tree node consists of three pieces of data: a *label*, a *value* and an ordered list of *children*, each of them a tree by themselves.
- the labels for tree nodes are either words not containing a slash `/` or the special symbol $\triangleright$; in the implementation $\triangleright$ corresponds to `NULL`. The latter is used to indicate that an entry in the tree corresponds to text that was deleted.
- the children of a tree node form a list of subtrees, i.e. are ordered. In addition, several subtrees in such a list may use the same label. This makes it possible to accommodate concrete files where entries that are logically connected are stored scattered between unrelated entries like the `AcceptEnv` entries in `sshd_config`.

We write $k = v \mapsto t$ for a tree node with label k, value v and children t. If it is clear from the context, or unimportant, v will often be omitted.

1.1. **Tree labels.** We take the tree labels from the set of *path components* $\mathcal{K} = (\Sigma \setminus \{/\})^+ \cup \{\triangleright\}$, that is, a tree label is any word not containing a backslash or the special symbol $\triangleright$. For tree labels, we define a partial concatenation operator $\odot$, as

$$k_1 \odot k_2 = \begin{cases} k_1 & \text{if } k_2 = \triangleright \\ k_2 & \text{if } k_1 = \triangleright \\ \text{undefined} & \text{otherwise} \end{cases}$$

Defining tree labels in this way (1) guarantees that there is a one–to–one correspondence between a tree label and the word it came from in the concrete text and (2) avoids any pain in splitting tree labels in the *put* direction.

1.2. **Trees.** The set of *ordered trees* $\mathcal{T}$ over $\Sigma^*_\triangleright$ is recursively defined as

- The empty tree $\triangleright$
- For any words $k \in \mathcal{K}$, $v \in \Sigma^*_\triangleright$ and any tree $t \in \mathcal{T}$, $[k = v \mapsto t]$ is in $\mathcal{T}$
- For any n and trees $[k_i = v_i \mapsto t_i] \in Trees$, the list $[k_1 = v_1 \mapsto t_1; k_2 = v_2 \mapsto t_2; \ldots; k_n = v_n \mapsto t_n]$ is in $\mathcal{T}$

Note that this allows the same key to be used multiple times in a tree; for example, $[\mathtt{a} \mapsto \mathtt{x}; \mathtt{a} \mapsto \mathtt{x}]$ is a valid tree and different from $[\mathtt{a} \mapsto \mathtt{x}]$.

The domain of a tree $\mathrm{dom}(t)$ is the list of all its labels, i.e. an element of $\mathtt{List}(\mathcal{K})$; for a tree $t = [k_1 \mapsto t_1; \ldots; k_n \mapsto t_n]$, $\mathrm{dom}(t) = [k_1; \ldots; k_n]$.

The concatenation of trees $t_1 \cdot t_2$ is simply list concatenation.

For sets $K \subset \mathcal{K}$ and $T \subset \mathcal{T}$, $[K \mapsto T]$ denotes the set of all trees $t = [k \mapsto t']$ with $k \in K$, $t' \in T$.

1.3. **Concatenation and iteration.** For a tree $t \in \mathcal{T}$, we define its underlying *key language* $\kappa(t)$ by

$$\kappa(t) = \begin{cases} / & \text{if } t = \rhd \text{ or } t = [\rhd \mapsto t_1] \\ k \cdot / & \text{if } t = [k \mapsto t_1] \\ k \cdot / \cdot \kappa(t_2) & \text{for } t = [k \mapsto t_1; t_2] \end{cases}$$

where $k_1 \cdot k_2$ is normal string concatenation. The key language of a set of trees $\kappa(T)$ is defined as $\{\kappa(t) | t \in T\}$.

In analogy to languages, we call two tree sets $T_1, T_2 \subseteq \mathcal{T}$ *unambiguously concatenable* if the key languages $\kappa(T_1)$ and $\kappa(T_2)$ are unambiguously concatenable. A tree set $T \in \mathcal{T}$ is *unambiguously iterable* if the underlying key language $\kappa(T)$ is unambiguously iterable.

1.4. **Public tree operations.** We need the public API to support the following operations. The set $P \subseteq \Sigma^*$ are paths

- $\mathit{lookup}(p, t) : P \times \mathcal{T} \longrightarrow \mathcal{T}$ finds the tree with path p
- $\mathit{assign}(p, v, t) : P \times \Sigma^* \times \mathcal{T} \longrightarrow \mathcal{T}$ assigns the value v to the tree node p
- $\mathit{remove}(p, t) : P \times \mathcal{T} \longrightarrow \mathcal{T}$ removes the subtree denoted by p
- $\mathit{get}(p, t) : P \times \mathcal{T} \longrightarrow \Sigma^*$ looks up the value associated with p
- $\mathit{ls}(p, t) : P \times \mathcal{T} \longrightarrow \mathtt{List}(\mathcal{T})$ lists all the subtrees underneath p

2. Lenses

Lenses map between strings in the regular language C and trees $T \subseteq \mathcal{T}$. They can also produce keys from a regular language K; these keys are used by the *subtree* lens to construct new trees.

A lens l consists of the functions *get*, *put*, *create*, and *parse*.

Lenses here are written as $l : C \overset{K,S,L}{\Longleftrightarrow} T$ where K and C are regular languages and $T \subseteq \mathcal{T}$. The skeletons $S \subseteq \mathcal{S}$ and dictionary type specifications L are as for Boomerang (really ??) Intuitively, the notation says that l is a lens that takes strings from C and transforms them to trees in T. Generally,

$$\frac{C \subseteq \Sigma_{\triangleright}^* \qquad K \subseteq \texttt{List}(\mathcal{K}) \qquad T \subseteq \mathcal{T} \qquad S \subseteq \mathcal{S} \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S}))}{l \in C \overset{K,S,L}{\Longleftrightarrow} T}$$

$$\begin{aligned} \mathit{get} &\in C \longrightarrow T \\ \mathit{parse} &\in C \longrightarrow K \times S \times D(L) \\ \mathit{put} &\in T \longrightarrow K \times S \times D(L) \longrightarrow C \times D(L) \\ \mathit{create} &\in T \longrightarrow K \times D(L) \longrightarrow C \times D(L) \end{aligned}$$

2.1. **const.** The $\mathit{const}\, E\, t\, v$ maps words matching E in the *get* direction to a fixed tree t and maps that fixed tree t back in the *put* direction. When text needs to be created from t, it produces the default word v.

$$\frac{E \in \mathcal{R} \qquad t \in T \qquad u \in [\![E]\!] \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S}))}{\mathit{const}\, E\, t\, u \in [\![E]\!] \overset{\triangleright,[\![E]\!],L}{\Longleftrightarrow} \{t\}}$$

$$\begin{aligned} \mathit{get}\, c &= t \\ \mathit{parse}\, c &= \triangleright, c, \{\} \\ \mathit{put}\, t\, (k, s, d) &= s, d \\ \mathit{create}\, t\, (k, d) &= u, d \end{aligned}$$

The *del* lens is syntactic sugar: $\mathit{del}\, E\, u = \mathit{const}\, E\, []\, u$.

2.2. **copy.** Copies a word into a leaf.

$$\frac{E \in \mathcal{R} \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S}))}{\mathit{copy} \in [\![E]\!] \overset{\triangleright,[\![E]\!],L}{\Longleftrightarrow} [[\![E]\!]]}$$

$$\begin{aligned} \mathit{get}\, c &= [c] \\ \mathit{parse}\, c &= \triangleright, c, \{\} \\ \mathit{put}\, [v]\, (k, s, d) &= v, d \\ \mathit{create}\, [v]\, (k, d) &= v, d \end{aligned}$$

2.3. **seq.** Gets the next value from a sequence as the key. We assume there's a generator $\texttt{nextval} : \Sigma^* \to \mathbb{N}$ that returns successive numbers on each invocation. `D` is the regular expression `[0-9]+` that matches positive numbers.

$$\frac{w \in \Sigma^* \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S})) \qquad n = \texttt{nextval}(w)}{\mathit{seq}\, w \in \epsilon \overset{[\![\texttt{D}]\!],\epsilon,L}{\Longleftrightarrow} []}$$

$$\begin{aligned} \mathit{get}\, \epsilon &= [] \\ \mathit{parse}\, \epsilon &= n, \epsilon, \{\} \\ \mathit{put}\, []\, (k, \epsilon, d) &= \epsilon, d \\ \mathit{create}\, []\, (k, d) &= \epsilon, d \end{aligned}$$

2.4. **label.** Uses a fixed tree label

$$\frac{w \in \Sigma^* \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S}))}{\mathit{label}\, w \in \epsilon \overset{w,\epsilon,L}{\Longleftrightarrow} []}$$

$$\begin{aligned} \mathit{get}\, \epsilon &= [] \\ \mathit{parse}\, \epsilon &= w, \epsilon, \{\} \\ \mathit{put}\, []\, (w, \epsilon, d) &= \epsilon, d \\ \mathit{create}\, []\, (k, d) &= \epsilon, d \end{aligned}$$

2.5. **key.** Uses a parsed tree label

$$\frac{E \in \mathcal{R} \qquad L \in \texttt{List}(\mathcal{P}(\mathcal{S}))}{\mathit{key}\, E \in [\![E]\!] \overset{[\![E]\!],\epsilon,L}{\Longleftrightarrow} []}$$

$$\begin{aligned} \mathit{get}\, c &= [] \\ \mathit{parse}\, c &= c, \epsilon, \{\} \\ \mathit{put}\, []\, (c, \epsilon, d) &= c, d \\ \mathit{create}\, []\, (c, d) &= c, d \end{aligned}$$

2.6. **subtree.** The subtree combinator $[l]$ constructs a subtree from l

$$\frac{l \in C \overset{K,S,L}{\Longleftrightarrow} T}{[l] \in C \overset{\triangleright,\square,S::L}{\Longleftrightarrow} [K \mapsto T]}$$

$$\begin{aligned}
\mathit{get}\, c &= [l.\, \mathit{key}\, c \mapsto l.\, \mathit{get}\, c] \\
\mathit{parse}\, c &= \triangleright, \square, \{l.\, \mathit{key}\, c \mapsto [l.\, \mathit{parse}\, c]\} \\
\mathit{put}\, [k \mapsto t]\, (k', \square, d) &= \begin{cases} \pi_1 \left(l.\, \mathit{put}\, t\, (k, \bar{s}, \bar{d})\right), d' & \text{if } (\bar{k}, \bar{s}, \bar{d}), d' = \mathit{lookup}(k, d) \\ \pi_1 \left(l.\, \mathit{create}\, t\, (k, \{\})\right), d & \text{if } \mathit{lookup}(k, d) \text{ undefined} \end{cases} \\
\mathit{create}\, [k \mapsto t]\, (k', d) &= \mathit{put}\, [k \mapsto t]\, (k', \square, d)
\end{aligned}$$

We store a triple (k, s, d) in dictionaries, but we don't use the stored key k.

2.7. **concat.** The concat combinator $l_1 \cdot l_2$ joins two trees.

$$\frac{l_1 \in C_1 \overset{K_1,S_1,L}{\Longleftrightarrow} T_1 \qquad l_2 \in C_2 \overset{K_2,S_2,L}{\Longleftrightarrow} T_2 \qquad C_1 \cdot^! C_2 \qquad \kappa(T_1) \cdot^! \kappa(T_2)}{l_1 \cdot l_2 \in C_1 \cdot C_2 \overset{K_1 \cdot K_2, S_1 \times S_2, L}{\Longleftrightarrow} T_1 \cdot T_2}$$

$$\begin{aligned}
\mathit{get}\, (c_1 \cdot c_2) &= (l_1.\, \mathit{get}\, c_1) \cdot (l_2.\, \mathit{get}\, c_2) \\
\mathit{parse}\, c_1 \cdot c_2 &= k_1 \odot k_2, (s_1, s_2), d_1 \oplus d_2 \\
\mathit{put}\, t_1 \cdot t_2\, (k, (s_1, s_2), d_1) &= c_1 \cdot c_2, d_3 \\
&\quad \text{where } c_i, d_{i+1} = l_i.\, \mathit{put}\, t_i\, (k, s_i, d_i) \\
\mathit{create}\, t_1 \cdot t_2\, (k, d_1) &= c_1 \cdot c_2, d_3 \\
&\quad \text{where } c_i, d_{i+1} = l_i.\, \mathit{create}\, t_i\, (k, d_i)
\end{aligned}$$

2.8. **union.** The union combinator $l_1 \,|\, l_2$ chooses.

$$\frac{l_i \in C_i \overset{K_i,S_i,L}{\Longleftrightarrow} T_i \text{ for } i = 1, 2 \qquad C_1 \cap C_2 = \emptyset \qquad S_1 \cap S_2 = \emptyset \qquad \kappa(T_1) \cap \kappa(T_2) = \emptyset}{l_1 \,|\, l_2 \in C_1 \cup C_2 \overset{K_1 \cup K_2, S_1 \cup S_2, L}{\Longleftrightarrow} T_1 \cup T_2}$$

$$get\, c = \begin{cases} l_1.\, get\, c & \text{if } c \in C_1 \\ l_2.\, get\, c & \text{if } c \in C_2 \end{cases}$$

$$parse\, c = \begin{cases} l_1.\, parse\, c & \text{if } c \in C_1 \\ l_2.\, parse\, c & \text{if } c \in C_2 \end{cases}$$

$$put\, t\, (k, s, d) = \begin{cases} l_1.\, put\, t\, (k, s, d) & \text{if } t, s \in T_1 \times S_1 \\ l_2.\, put\, t\, (k, s, d) & \text{if } t, s \in T_2 \times S_2 \\ l_1.\, create\, t\, (k, d) & \text{if } t, s \in (T_1 \setminus T_2) \times S_2 \\ l_2.\, create\, t\, (k, d) & \text{if } t, s \in (T_2 \setminus T_1) \times S_1 \end{cases}$$

$$create\, t\, (k, d) = \begin{cases} l_1.\, create\, t\, (k, d) & \text{if } t \in T_1 \\ l_2.\, create\, t\, (k, d) & \text{if } t \in T_2 \setminus T_1 \end{cases}$$

2.9. **star.** The star combinator l^* iterates.

$$\frac{l \in C \overset{K}{\Longleftrightarrow} T \qquad C^{!*} \qquad \kappa(T)^{!*}}{l^* \in C^* \overset{K^*}{\Longleftrightarrow} T^*}$$

$$get\, c_1 \cdots c_n = (l.\, get\, c_1) \cdots (l.\, get\, c_n)$$

$$parse\, c_1 \cdots c_n = k_1 \odot \ldots \odot k_n, [s_1; \ldots; s_n], d_1 \oplus \ldots \oplus d_n$$
$$\text{where } k_i, s_i, d_i = l.\, parse\, c_i$$

$$put\, t_1 \cdots t_n\, (k, [s_1; \cdots; s_m], d_1) = (c_1 \cdots c_n), d_{n+1}$$
$$\text{where } c_i, d_{i+1} = \begin{cases} l.\, put\, t_i\, (k, s_i, d_i) & \text{for } 1 \le i \le \min(m, n) \\ l.\, create\, t_i\, (k, d_i) & m + 1 \le i \le n \end{cases}$$

$$create\, t_1 \cdots t_n\, (k, d_1) = (c_1 \cdots c_n), d_{n+1}$$
$$\text{where } c_i, d_{i+1} = l.\, create\, t_i\, (k, d_i)$$

Want reordering and insertion in the middle to be reflected. If $get\, c_1 \cdot c_2 = t_1 \cdot t_2$, want $put\, (t_2 \cdot t_1)\, (c_1 \cdot c_2) = l.\, put\, t_2\, (c_2) \cdot l.\, put\, t_1\, (c_1)$ This can only happen if the information to be reordered is in subtrees. In particular, comment lines need to become their own subtree, with some support from the language to create 'hidden' entries. Simplest: allow `NULL` as the key for a subtree and ignore such tree entries in the public API.

Need to split a tree $t \in T$ into subtrees according to ?? Keeping a fake 'slot' $\triangleright$ around for text that didn't produce a tree should help with that.

For K^* to make any sense, must have $\triangleright \in K$ and the application of $(l^*).\,parse$ must return $\triangleright$ for all except at most one application.

3. Regular expressions and languages

For type checking, we need to compute the following properties of regular languages R, R_1, R_2

- decide unambiguous concatenation $R_1 \cdot^! R_2$ and compute $R_1 \cdot R_2$
- decide unambiguous iteration $R^{!*}$ and compute R^*
- disjointness $R_1 \cap R_2 = \emptyset$ (we don't need general intersection, though I don't know of a quicker way to decide disjointness)
- compute the regular language $R = [\![E]\!]$ for a regular expression $E \in \mathcal{R}$