Binary Search Tree Analysis

Ido Begun, 200397834
Yael Harel, 301286332
Idan Shabat, 203511597

Thursday 15$^{\text{th}}$ September, 2016

Abstract

Binary Search Trees (abbr. BST) are a useful database for storing data, which its elements can be compared one to another (e.g. integers or strings). Programs which use BSTs should support operations like insertions, deletions and creations of new nodes in the tree, while keeping the BST structure valid. This paper will introduce an analysis of such programs, where the purpose is to verify the validity of the BSTs that were defined in the program.

1 Introduction

1.1 BST Definition

A binary search tree is a rooted tree, where each of its nodes has at most two children: "left" and "right". Also, each node contains a key, which in our case would be an integer, such that the keys satisfy the following rule: given a node n with a key d,

1. for each key d' of a node in n's left subtree, $d' \leq d$
2. for each key d'' of a node in n's right subtree, $d'' \geq d$

1.2 BST Implementation

For analyzing actual programs, we will assume that each node of the BST is represented by a structure (or class) with 3 fields: *data, left, right*. *data* is the key stored in the node, and *left* and *right* are the left and right children, respectively.

The analyzing method will consider programs such that each statement in it is one of the regular statements: $if, while$, or one of the commands in table 1.

Command	Effect
$createNode(n, d)$	Creates a new node n with the key d
$setLeft(n, n')$	Sets the left child of n to n'
$setRight(n, n')$	Sets the right child of n to n'
$setValue(n, d)$	Sets the key of the node n to d
$setValue(n, n + d)$	increment the key of the node n by d
$setValue(n, n - d)$	decrement the key of the node n by d

Table 1: Commands table.

2 Lattices and Galois Connection

2.1 Concrete States Lattice

The state at each point of the analyzed program will be considered as a collection of nodes and the values of their fields. Namely, if N is the set of the nodes created in the program, then a state is a **partial** function $s : N \to N \times \mathbb{Z} \times N$, such that $s(n) = (n.left, n.data, n.right)$. I.e. s is an univalent relation and if $s(n)$ isn't defined, we denote it $s(n) = \#$. Actually, $n.left$ or $n.right$ may be $null$, so we should consider the set $N^+ = N \cup \{null\}$, so a state is $s : N \to N^+ \times \mathbb{Z} \times N^+$.

In a single point in the program, there might be several states that are possible (depends on the inputs). Therefore, we represent each point as a **collection** of states $S \subseteq \{s | s : N \to N^+ \times \mathbb{Z} \times N^+\}$. Also, we have to ensure that if $s(n) = \#$ (n isn't defined), then there's no $n' \in N$ such that $n'.left = n$ or $n'.right = n$. We will also need a special element for when we cannot delimit the possible states - this element is denoted by $\top$.

Finally, the lattice C of the concrete states is defined by:

$$C = P_{<\infty}(\{s \mid s \text{ is a partial function } s : N \to N^+ \times \mathbb{Z} \times N^+$$

$$\text{s.t. } s(n) = (n_1, d, n_2) \implies s(n_1), s(n_2) \neq \#\}) \cup \{\top\},$$

where for a set D, $P_{<\infty}(D)$ is the collection of the finite subsets of D. The properties of this lattice are:

$$S \sqsubseteq T \iff S \subseteq T$$

$$S \sqcup T := S \cup T$$

$$S \sqcap T := S \cap T$$

For $\top$, every $S \in C$ satisfy $S \sqsubseteq \top$, $S \sqcup \top = \top \sqcup S = \top$ and $S \sqcap \top = \top \sqcap S = S$.

It is easy to check that those definitions create a complete lattice.

2.2 Abstract States Lattice

As the purpose of the analysis is to check the validity of the BST, we would like to define a representation of the concrete states so that it will be easy to check the order of the keys. A BST will be represented by a string that indicates the structure of the BST and the range of the possible values of each node.

Formally, a ***BST string*** will be defined recursively:

1. The empty string, ε, is a *BST string*.
2. If $n \in N, d_1, d_2 \in \mathbb{Z}$ and r_1, r_2 are *BST strings*, then $(r_1 n[d_1 - d_2] r_2)$ is also a *BST string*.

Intuitively, the *BST string* $\omega = (r_1 n[d_1 - d_2] r_2)$ represents a BST which its root is the node n with the data between d_1 and d_2, its left subtree is represented by r_1 and its right subtree is represented by r_2. We will later see that sometimes we would like to express the idea that **we don't know** the identity of the node in a certain place in the tree. Therefore, in the *BST string* $(r_1 n[d_1 - d_2] r_2)$, we allow n to be n_τ - a special node that was designed for this ($n_\tau \notin N$, and the programmer can't access it).

By T we denote the set of the *BST strings*. Now, the lattice A of the abstract states is defined by:

$$A = \{X \subseteq T \mid \forall \omega \in X : \mathit{root}(\omega) \notin \{null, n_\tau\},$$

$$\forall \omega_1, \omega_2 \in X : \mathit{root}(\omega_1) = \mathit{root}(\omega_2) \implies \omega_1 = \omega_2\} \cup \{\tau\}$$

where *root* of a *BST string* defined as follows:

$$\mathit{root}((r_1 n[d_1 - d_2] r_2)) = n; \ \mathit{root}(\varepsilon) = null$$

And τ is an additional special abstract element, which expresses the idea that we don't know the structure or the keys range in a concrete state.

The reason that we keep **a subset** of *BST strings* is that there might be more than one BST at the same time in the analyzed program. We can see that two different *BST strings* in X cannot have the same root, and that the empty string and n_τ-rooted strings cannot be in X.

Example 1. *A valid abstract state:*
$\{\Big((n_1[0-3])n_0[4-5]\Big), \Big(n_2[0-0](n_1[0-3])\Big)\}$. *The first string has the root n_0, and the second's root is n_2.*

We now define the partial order over the set. First, we recursively define an order over *BST strings*, then over the elements of A:

1. For any $\omega \in T$, $\varepsilon \preceq \omega$,
2. $(r_1 n[d_1 - d_2]r_2) \preceq (r'_1 n'[d'_1 - d'_2]r'_2)$ iff:
   - $r_1 \preceq r'_1$, $r_2 \preceq r'_2$
   - $d'_1 \leq d_1$, $d_2 \leq d'_2$ (i.e. $[d_1, d_2] \subseteq [d'_1, d'_2]$)
   - $n = n'$ or $n' = n_\tau$
3. $\forall X \in A,\ X \sqsubseteq \tau$
4. For $X, Y \in A \setminus \{\tau\},\ X \sqsubseteq Y \iff \forall \omega_1 \in X\ \exists \omega_2 \in Y\ s.t.\ \omega_1 \preceq \omega_2$

A simple inductive proof would show that $\preceq$ is a partial order over the set T, and after that it is easy to verify that $\sqsubseteq$ is reflexive and transitive. The proof of its antisymmetry is the following:

Assume that $X \sqsubseteq Y$ and $Y \sqsubseteq X$.

If $X = \tau$ or $Y = \tau$, then by $\sqsubseteq$ definition: $X = Y = \tau$.

Else, let $\omega \in X$. There is some $\omega' \in Y$ s.t. $\omega \preceq \omega'$, and there is $\omega'' \in X$ s.t. $\omega' \preceq \omega''$. Therefore, we get $\omega \preceq \omega''$. Note that $\omega \neq \varepsilon$ and that ω, ω'' can't be with the root n_τ, so by the definition of $\preceq$: $root(\omega) = root(\omega'')$. But then $\omega = \omega''$ (X cannot contain two different strings with the same root). In conclusion, we have $\omega \preceq \omega' \preceq \omega$ and that implies $\omega = \omega' \in Y$. We proved that $X \subseteq Y$, and by symmetry: $X = Y$.

$\square$

Before the definitions of the $\sqcup$ and $\sqcap$ operators, we define a helper function:

$$treeof_X(n) = \begin{cases} \omega & \exists \omega \in X \text{ s.t. } root(\omega) = n \\ \varepsilon & \text{else} \end{cases}$$

where $X \in A \setminus \{\tau\}$ and $n \in N$. Given a set $X \in A \setminus \{\tau\}$, this function returns the unique *BST string* in X with the specified node as its root. If such string doesn't exists, ε will be returned.

Join and *meet* will also be defined first to *BST strings* and then to A:

1. For any $\omega \in T$, $\varepsilon \vee \omega = \omega \vee \varepsilon = \omega$
2. $(r_1 n[d_1 - d_2]r_2) \vee (r'_1 n[d'_1 - d'_2]r'_2) =$
   $(r_1 \vee r'_1 n[min\{d_1, d'_1\}, max\{d_2, d'_2\}]r_2 \vee r'_2)$
3. If $n \neq n'$, $(r_1 n[d_1 - d_2]r_2) \vee (r'_1 n'[d'_1 - d'_2]r'_2) =$
   $(r_1 \vee r'_1 n_\tau[min\{d_1, d'_1\}, max\{d_2, d'_2\}]r_2 \vee r'_2)$
4. $\forall X \in A,\ \tau \sqcup X = X \sqcup \tau = \tau$
5. For $X, Y \in A \setminus \{\tau\},\ X \sqcup Y = \{treeof_X(n) \vee treeof_Y(n) \mid n \in N\}$
6. For any $\omega \in T$, $\varepsilon \wedge \omega = \omega \wedge \varepsilon = \varepsilon$
7. $(r_1 n[d_1 - d_2]r_2) \wedge (r'_1 n[d'_1 - d'_2]r'_2) =$
   $(r_1 \wedge r'_1 n[max\{d_1, d'_1\}, min\{d_2, d'_2\}]r_2 \wedge r'_2)$
8. If $n \neq n'$ and $n, n' \neq n_\tau$, $(r_1 n[d_1 - d_2]r_2) \wedge (r'_1 n'[d'_1 - d'_2]r'_2) = \varepsilon$
9. If $n \neq n_\tau$, $(r_1 n[d_1 - d_2]r_2) \wedge (r'_1 n_\tau[d'_1 - d'_2]r'_2) =$
   $(r_1 \wedge r'_1 n[max\{d_1, d'_1\}, min\{d_2, d'_2\}]r_2 \wedge r'_2)$
10. $\forall X \in A,\ \tau \sqcap X = X \sqcap \tau = X$

11. For $X, Y \in A \setminus \{\tau\}$, $X \sqcap Y = \{treeof_X(n) \wedge treeof_Y(n) \mid n \in N\}$

Example 2. $\Big((n_1[2-3])n_0[4-6](n_2[8-8])\Big) \vee \Big((n_1[0-0])n_0[3-5](n_2[6-9])\Big) = \Big((n_1[0-3])n_0[3-6](n_2[6-9])\Big)$,

$\Big((n_1[2-3])n_0[4-6](n_3[8-9])\Big) \vee \Big(n_0[3-5](n_2[6-9])\Big) =$
$\Big((n_1[2-3])n_0[3-6](n_\tau[6-9])\Big)$.

2.3 The Galois Connection

In order to formalize the idea of the *BST strings* as representations of collections of BSTs, we will now define a Galois connection (C, α, γ, A) between the lattices. Recall that $\alpha : C \to A$ is the abstraction function (maps concrete states to their abstract representations) and $\gamma : A \to C$ is the concretization function (translate the abstract representation into a concrete state).

2.3.1 The Abstraction Function α

Before we can define α, we will define some terms and functions.

1. A ***cycle*** in a state $s : N \to N^+ \times \mathbb{Z} \times N^+$ is a sequence of nodes $n_1, n_2, \ldots, n_t$ such that $\forall i = 1, \ldots, t-1$, $s(n_i) = (n_{i+1}, ...)$ or $s(n_i) = (..., n_{i+1})$, and $s(n_t) = (n_1, ...)$ or $s(n_t) = (..., n_1)$. i.e. each node is a parent of the next one.

2. A ***root node***, with respect to a state $s : N \to N^+ \times \mathbb{Z} \times N^+$, is a node $n \in N$ such that $s(n) \neq \#$ and $\forall n' \in N, s(n') \neq (n, ...)$ and $s(n') \neq (..., n)$. Namely, n has no parents in s. For $m \in N$ a set of states S, we denote: $G_m(S) = \{s \in S \mid m$ is a *root node* of $s\}$.

3. Given a state $s : N \to N^+ \times \mathbb{Z} \times N^+$ and a node (or null) $n \in N^+$, we define recursively the function $down_s(n)$:

   - $down_s(null) = \varepsilon$
   - For $n \neq null$ *s.t.* $s(n) = (n_1, d, n_2)$,
     $down_s(n) = (down_s(s(n_1))n[d-d]down_s(s(n_2))$

   This function creates the matched *BST string* of the subtree of n. Note that $down_s(n)$ isn't defined only if s has a *cycle*, and indeed, we use it only for s's without *cycles*.

   Example 3. *Let s be defined by:*
   $s(n_0) = (n_1, 4, n_3)$,
   $s(n_1) = (null, 1, n_2)$,
   $s(n_2) = (null, 3, null)$,
   $s(n_3) = (null, 6, null)$.
   Then: $down_s(n_0) = (down_s(n_1)n_0[4-4]down_s(n_3)) =$
   $\Big((n_1[1-1](n_2[3-3]))n_0[4-4](n_3[6-6])\Big)$.

4. Given a state $S \in C$, if $S = \top$ we define $\alpha(S) = \tau$. Else,

$$\alpha(S) = \begin{cases} \tau & S \text{ contains a state with a } \textit{cycle} \\ \bigsqcup\limits_{s \in S} \{down_s(n) \mid n \text{ is a } \textit{root node} \text{ in } s\} & \textit{else} \end{cases}$$

   As it can be seen, if S contains a state with a *cycle*, α will return τ - means that we cannot interpret it as a tree at all. Else, we take all the *root nodes* in the states of S and create their *BST string*. Then we collect all the created *BST strings* and join them (the *join* definition can be naturally extended to any finite collection of sets).
   A useful fact is that by $\sqcup$ definition, in case that $S \neq \top$ and doesn't contain a state with a *cycle*, $\alpha(S)$ can also be written as:

$$\alpha(S) = \{ \bigvee_{s \in G_m(S)} down_s(m) \mid m \in N\}$$

Lemma 1. *Let s be a state. For every node n' s.t. $s(n') \neq \#$, there exists a root node n of s, s.t. $down_s(n')$ is a substring of $down_s(n)$.*

Proof. By looking for a *parent* of n' (i.e. n'' s.t. $s(n'') = (n', ...) or s(n'') = (..., n')$), then looking for a *parent* of its *parent*, and so on, we can find a sequence of nodes: $n_0, n_1, n_2, ..., n_k$ such that:

- n_k is a *root node* of s,
- $n_0 = n'$,
- $\forall i = 1, ..., k,\ n_i$ is a *parent* of n_{i-1}.

Now we can prove by induction that $down_s(n')$ is a substring of $down_s(n_i)$, for all i.

- For $i = 0$, $down_s(n_0) = down_s(n')$ and we're done.
- For $i > 0$, w.l.o.g. we assume that $s(n_i) = (n_{i-1}, ...)$. Then, $down_s(n_i) = (down_s(n_{i-1}, ...)$, and since $down_s(n_{i-1}$ has $down_s(n')$ as a substring, so is $down_s(n_i)$.

In conclusion, we get that $n = n_k$ is a *root node* of s, and $down_s(n') = down_s(n_0)$ is a substring of $down_s(n) = down_s(n_k)$. □

2.3.2 The Concretization Function γ

The definition of γ can completely derived from α's definition, as known from the general theory of Galois Connections. We now write this definition, using the terminology that already mentioned:

For $X \in A$:

$$\gamma(X) = \begin{cases} \top & X = \tau \\ \{s \mid s \text{ doesn't contain a } \textit{cycle} \text{ and } \forall \textit{root node } n \text{ in } s\ \exists \omega \in X : down_s(n) \preceq \omega\} & \text{else} \end{cases}$$

In words, every BST that we can build from a state $s \in \gamma(X)$ has to be represented by a (part of) $\omega \in X$.

2.3.3 The Galois Connection Property

To show that (C, α, γ, A) is indeed a Galois connection, the following property should be proved:

$$\forall S \in C \text{ and } X \in A,\ \alpha(S) \sqsubseteq X \iff S \sqsubseteq \gamma(X)$$

We prove the property by proving each implication separately:

- $\Rightarrow$:
  If $S = \top$ or S contains a state with a *cycle*, then $\alpha(S) = \tau$, and therefore we know that $\tau \sqsubseteq X$. It is only possible if $X = \tau$, and that means that $\gamma(X) = \top$, so obviously $S \sqsubseteq \top = \gamma(X)$.
  Else, S is a set of states which none of them contains a *cycle*. Given a $s \in S$, we have to show that for every *root node* n, there is $\omega \in X$ s.t. $down_s(n) \preceq \omega$.
  Let $n \in N$ be a *root node* in s. $down_s(n)$ is well defined because s doesn't contain cycles. When applying α on S, $down_s(n)$ is joined with other *BST strings*, and the result ω' satisfy $down_s(n) \preceq \omega' \preceq \omega$ for some $\omega \in X$ (because it is given that $\alpha(S) \sqsubseteq X$). By transitivity of $\preceq$, $down_s(n) \preceq \omega$, and we get that $s \in \gamma(X)$.

- $\Leftarrow$:
  If $X = \tau$, then $\alpha(S) \sqsubseteq X$ is trivially satisfied.
  If $S = \top$, then since $S \sqsubseteq \gamma(X)$, it has to be that $\gamma(X) = \top$. That means that $X = \tau$, and of course $\alpha(S) \sqsubseteq X$.
  If S contains a state with a *cycle*, then it's impossible that $X \neq \tau$, because otherwise we have $S \subseteq \gamma(X)$ and $\gamma(X)$ is a set of states **without** *cycles*. Therefore, $X = \tau$, and again we get instantly that $\alpha(S) \sqsubseteq X$.
  Else, fix $n \in N$. For every $s \in G_n(S)$, we know that $s \in \gamma(X)$ and therefore there exists some $\omega_s \in X$ with $down_s(n) \preceq \omega_s$. By $\preceq$ definition, the root of each of these ω_s is n, and therefore they have to be **the same *BST string*** (because X doesn't contain two different strings with the same root). So, we can denote $\omega_s = \omega$, and then for every $s \in G_n(S)$:

$down_s(n) \preceq \omega \implies \bigvee_{s \in G_n(S)} down_s(n) \preceq \omega.$
This is true for all $n \in N$, and therefore:

$$\alpha(S) = \{ \bigvee_{s \in G_n(S)} down_s(n) \mid n \in N\} \sqsubseteq X$$

$\square$

3 Transformers

The last step before we can continue to analyzing actual BST programs is to decide how we treat the commands in the program. The commands will act as a transformers on the Abstract lattice: $f^{\#} : A \to A$. The transformers will be tested with respect to f - the effect of the commands on the concrete states.

3.1 Replacement Functions

Most of the abstract transformers will use manipulations on *BST strings*, especially replacement of substrings with other strings. We now define three helper functions for replacing the left subtree of a node n, the right subtree of n or the value of n.
Given $\omega \in T$, $n \in N$ and $d \in \mathbb{Z}$, the functions $reL_{n,\omega}$, $reR_{n,\omega}$ and $reV_{n,d}$ will be defined recursively:

- $reL_{n,\omega}(\varepsilon) = reR_{n,\omega}(\varepsilon) = reV_{n,d}(\varepsilon) = \varepsilon$
- If $n' \notin \{n, n_\tau\}$,
  $reL_{n,\omega}((r_1 n'[d_1 - d_2] r_2)) = (reL_{n,\omega}(r_1) n'[d_1 - d_2] reL_{n,\omega}(r_2))$
  $reR_{n,\omega}((r_1 n'[d_1 - d_2] r_2)) = (reR_{n,\omega}(r_1) n'[d_1 - d_2] reR_{n,\omega}(r_2))$
  $reV_{n,d}((r_1 n'[d_1 - d_2] r_2)) = (reV_{n,d}(r_1) n'[d_1 - d_2] reV_{n,d}(r_2))$
- For n,
  $reL_{n,\omega}((r_1 n[d_1 - d_2] r_2)) = (\omega \vee reL_{n,\omega}(r_1) n[d_1 - d_2] reL_{n,\omega}(r_2))$
  $reR_{n,\omega}((r_1 n[d_1 - d_2] r_2)) = (reR_{n,\omega}(r_1) n[d_1 - d_2] \omega \vee reR_{n,\omega}(r_2))$
  $reV_{n,d}((r_1 n[d_1 - d_2] r_2)) = (reV_{n,d}(r_1) n[d - d] reV_{n,d}(r_2))$
- For n_τ,
  $reL_{n,\omega}((r_1 n_\tau[d_1 - d_2] r_2)) = (\omega \vee reL_{n,\omega}(r_1) n_\tau[d_1 - d_2] reL_{n,\omega}(r_2))$
  $reR_{n,\omega}((r_1 n_\tau[d_1 - d_2] r_2)) = (reR_{n,\omega}(r_1) n_\tau[d_1 - d_2] \omega \vee reL_{n,\omega}(r_2))$
  $reV_{n,d}((r_1 n_\tau[d_1 - d_2] r_2)) = (reV_{n,d}(r_1) n_\tau[min\{d_1, d\} - max\{d_2, d\}] reV_{n,d}(r_2))$

As we can see, n_τ is treated like it may be n, but also may not.

Lemma 2. *$reL_{n,\omega}$, $reR_{n,\omega}$ and $reV_{n,d}$ are monotonic,
i.e. $\omega_1 \preceq \omega_2 \implies reL_{n,\omega}(\omega_1) \preceq reL_{n,\omega}(\omega_2)$, $reR_{n,\omega}(\omega_1) \preceq reR_{n,\omega}(\omega_2)$ and $reV_{n,d}(\omega_1) \preceq reV_{n,d}(\omega_2)$*

Lemma 3. *$reL_{n,\omega}(\omega_1) \vee reL_{n,\omega}(\omega_2) \preceq reL_{n,\omega}(\omega_1 \vee \omega_2)$,
$reR_{n,\omega}(\omega_1) \vee reR_{n,\omega}(\omega_2) \preceq reR_{n,\omega}(\omega_1 \vee \omega_2)$ and
$reV_{n,d}(\omega_1) \vee reV_{n,d}(\omega_2) \preceq reV_{n,d}(\omega_1 \vee \omega_2)$.*

The proof of these lemmas is by a simple structural induction, where the second one can be easily derived from the first.

3.2 Transformers Definitions

First of all, for every concrete transformer f and abstract transformer $f^{\#}$, $f(\top) = \top$, $f^{\#}(\tau) = \tau$. An addition concept that should be noticed in the definitions is that in some states, a command may be **not valid** (i.e. create a new node with a name that already exists etc.). In those cases, as it can be seen from the following definitions, we just ignore the "invalid" states. The following definitions are for the non-trivial cases. In all of the definitions, $s[n \leftarrow a]$ is the same state as s, only with the output of n set to a.

1. For each $S \in C,\ n \in N,\ d \in \mathbb{Z}:$

$$[\![createNode(n,d)]\!](S) = \{s[n \leftarrow (null, d, null)] \mid s \in S \text{ s.t. } s(n) = \#\}$$

2. For each $S \in C,\ n, n' \in N:$

$$[\![setLeft(n,n')]\!](S) =$$
$$\{s[n \leftarrow (n', d, n_r)] \mid s \in G_{n'}(S) \text{ s.t. } s(n) = (null, d, n_r)\} \setminus \{s' \mid s' \text{ is a state with a } cycle\}$$
$$[\![setRight(n,n')]\!](S) =$$
$$\{s[n \leftarrow (n_l, d, n')] \mid s \in G_{n'}(S) \text{ s.t. } s(n) = (n_l, d, null)\} \setminus \{s' \mid s' \text{ is a state with a } cycle\}$$

   Intuitively, the command $setLeft(n, n')$ makes sense only if n' is already defined as a *root node* of s, and n' is not one of the ancestors of n (therefore we subtract the states that after the change contain a cycle). Another restriction we added is that n can't have a left child (the same goes for $setRight(n, n')$).

3. For each $S \in C,\ n \in N,\ d \in \mathbb{Z}:$

$$[\![setValue(n,d)]\!](S) = \{s[n \leftarrow (n_l, d, n_r)] \mid s \in S \text{ s.t. } s(n) = (n_l, d', n_r)\}$$

We now define the matching abstract transformers:

1. For each $X \in A,\ n \in N,\ d \in \mathbb{Z}$,

$$[\![createNode(n,d)]\!]^{\#}(X) = X \sqcup \{(n[d-d])\}$$

2. First, for $X \in A,\ n, n' \in N$ we denote $\omega' = treeof_X(n')$, and then:

$$[\![setLeft(n,n')]\!]^{\#}(X) = \{reL_{n,\omega'}(\omega) \mid \omega \in X\} \setminus \{\omega'\}$$

   Symmetrically,

$$[\![setRight(n,n')]\!]^{\#}(X) = \{reR_{n,\omega'}(\omega) \mid \omega \in X\} \setminus \{\omega'\}$$

3. For each $X \in A,\ n \in N,\ d \in \mathbb{Z}$,

$$[\![setValue(n,d)]\!]^{\#}(X) = \{reV_{n,d}(\omega) \mid \omega \in X\}$$

3.3 Soundness Property

The following proofs will show that the abstract transformers defined above are **sound**, with respect to the matching concrete transformers. More precisely, we have to show that for every concrete transformer f, the matching abstract transformer $f^{\#}$, and $X \in A$:

$$\alpha(f(\gamma(X))) \sqsubseteq f^{\#}(X)$$

Note that for the edge case $X = \tau$, we get

$$\gamma(X) = \top \implies f(\gamma(X)) = \top \implies \alpha(f(\gamma(X))) = \tau$$

and indeed $f^{\#}(X) = \tau$.

Now we prove the soundness property for the non trivial cases.

1. $[\![createNode(n,d)]\!]^{\#}$ is **sound**:

   Proof. Suppose $\omega \in \alpha(f(\gamma(X)))$. Then, $\omega = \bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(m)$ for some $m \in N$. If $m = n$ then by f's definition, $\forall s' \in G_m(f(\gamma(X)))\ s'(n) = (null, d, null)$ and then $\forall s' \in G_m(f(\gamma(X)))\ down_{s'}(n) = (n[d-d]) \implies \omega = \bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(n) = (n[d-d]) \in f^{\#}(X)$.

   If $m \neq n$, then $down_{s'}(m) = down_s(m)$, where $s' = s[n \leftarrow (null, d, null)]$ and $s \in \gamma(X)$. This fact should be obvious, since s and s' have the same outputs on every node except n, and n can't appear in $down_s(m)$ or in $down_{s'}(m)$. Anyway, it can be simply proved by structural induction. So, we have: $\omega = \bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(m) = \bigvee_{s \in G_m((\gamma(X))} down_s(m) \in \alpha(\gamma(X))$.
   By the Galois connection property, we know that $\alpha(\gamma(X)) \sqsubseteq X$, i.e there is $\omega' \in X \subseteq X \cup (n[d-d])$ s.t. $\omega \preceq \omega'$.
   This gives us finally: $\alpha(f(\gamma(X))) \sqsubseteq f^{\#}(X)$.

   □

2. $[\![setLeft(n,n')]\!]^{\#}$ is **sound**:

   Proof. Here are two facts that will help us in the proof:

   - Say that $s' \in G_m(f(\gamma(X)))$ for some $m \in N$. In particular, $s' = s[n \leftarrow (n', d, n_r)]$ where $s \in \gamma(X)$ (and $s(n) = (null, d, n_r)$). Suppose now that m isn't a *root node* of s, and it has a parent m^*. If $m^* \neq n$, then s' has the same output on m^* as s, and therefore m cannot be a *root node* in s', in contradiction. It means that $m^* = n$, but then $s(m^*) = s(n) = (null, d, n_r)$ which means that $m = n_r$. But that is not possible either, since $s'(n) = (n', d, n_r) \implies m = n_r$ isn't a *root node* of s'. Therefore, **if m is a *root node* in s', then it is a *root node* of s.**
   - With the notations of the last paragraph, we now prove by structural induction that: $\forall m \in N^+\ down_{s'}(m) \preceq reL_{n,\omega'}(down_s(m))$ where $\omega' = treeof_X(n')$.
     - If $down_{s'}(m) = \varepsilon$, then $m = null$, so $reL_{n,\omega'}(down_s(m)) = reL_{n,\omega'}(\varepsilon) = \varepsilon$.
     - If $m \neq n$, and $s(m) = (m_1, d, m_2)$, then $s'(m) = (m_1, d, m_2)$, and by the induction assumption:
       $down_{s'}(m) = (down_{s'}(m_1)m[d-d]down_{s'}(m_2)) \preceq$
       $(reL_{n,\omega'}(down_s(m_1))m[d-d]reL_{n,\omega'}(down_s(m_2))) =$
       $reL_{n,\omega'}((down_s(m_1)m[d-d]down_s(m_2))) = reL_{n,\omega'}(down_s(m))$.
     - If $m = n$, and $s(n) = (null, d, n_r)$, then $s'(n) = (n', d, n_r)$, and therefore $down_{s'}(n) = (down_{s'}(n')n[d-d]down_{s'}(n_r))$. It has to be that $down_{s'}(n') = down_s(n')$, because the only possible difference between them is on n, which doesn't appear in $down_{s'}(n')$ (if it does, then s' contains a *cycle* in contradiction to f's definition). Remember that n' is a *root node* of s, and therefore (since $s \in \gamma(X)$) $down_s(n') \preceq treeof_X(n') = \omega'$.
       Finally, we got:
       $down_{s'}(n) = (down_{s'}(n')n[d-d]down_{s'}(n_r)) \preceq (\omega' n[d-d]reL_{n,\omega'}(down_s(n_r))) =$
       $(\omega' \vee reL_{n,\omega'}(\varepsilon)[d-d]reL_{n,\omega'}(down_s(n_r))) = reL_{n,\omega'}(down_s(n))$

   Now, if $\omega \in \alpha(f(\gamma(X)))$, then for some $m \in N$:
   $\omega = \bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(m) = \bigvee_{s \in G_m(\gamma(X))} down_{s'}(m) \preceq \bigvee_{s \in G_m(\gamma(X))} reL_{n,\omega'}(down_s(m)) \preceq$
   $reL_{n,\omega'}(\bigvee_{s \in G_m(\gamma(X))} down_s(m))$.
   In the last step we used lemma 3. Note that $\bigvee_{s \in G_m(\gamma(X))} down_s(m) \in \alpha(\gamma(X))$, and therefore $\bigvee_{s \in G_m(\gamma(X))} down_s(m) \preceq \omega''$ for some $\omega'' \in X$. Finally, by lemma 2 we get:
   $\omega \preceq reL_{n,\omega'}(\omega'') \in f^{\#}(X)$.

Actually, we have to show that $\omega \neq treeof_X(n')$. But otherwise, the root of $\omega = \bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(m)$ would be n'. It's impossible because $f(\gamma(X))$ doesn't contain states with n' as a root.

All that shows that $\alpha(f(\gamma(X))) \sqsubseteq f^\#(X)$.

$\square$

3. $[\![setRight(n, n')]\!]^\#$:

   Proof. Symmetric to $[\![setLeft(n, n')]\!]^\#$. $\square$

4. $[\![setValue(n, n')]\!]^\#$ is **sound**:

   Proof. Otherwise, we first prove the following statement by induction:
   Let $m \in N^+$ and $s \in \gamma(X)$ such that $s(n) = (n_1, d', n_2)$ and $s(m) \neq \#$. We denote $s' = s[n \leftarrow (n_1, d, n_2)]$. Then, $down_{s'}(m) = reV_{n,d}(down_s(m))$.

   - If $down_{s'}(m) = \varepsilon$ then $m = null$, which in that case: $down_s(m) = \varepsilon$, so $reV_{n,d}(down_s(m)) = \varepsilon$.
   - Else, $m \neq null$. If $m \neq n$:
     $down_{s'}(m) = (down_{s'}(m_1)m[d_m{-}d_m]down_{s'}(m_2))$, where $s'(m) = s(m) = (m_1, d_m, m_2)$.
     By induction,
     $(down_{s'}(m_1)m[d_m{-}d_m]down_{s'}(m_2)) = (reV_{n,d}(down_s(m_1))m[d_m{-}d_m]reV_{n,d}(down_s(m_2))) = reV_{n,d}((down_s(m_1)m[d_m - d_m]down_s(m_2))) = reV_{n,d}(down_s(m))$
   - If $m = n$: $down_{s'}(n) = (down_{s'}(n_1)n[d - d]down_{s'}(n_2)) =$
     $(reV_{n,d}(down_s(n_1))n[d - d]reV_{n,d}(down_s(n_2))) =$
     $reV_{n,d}((down_s(n_1)n[d' - d']down_s(n_2))) = reV_{n,d}(down_s(n))$.

   Each $\omega \in \alpha(f(\gamma(X)))$ is of the form $\bigvee_{s' \in G_m(f(\gamma(X)))} down_{s'}(m)$. From f's definition, it is clear that $s' \in G_m(f(\gamma(X))) \iff s' = s[n \leftarrow (n_1, d, n_2)]$ where $s \in G_m(\gamma(X))$ (Because f doesn't change the structure of the states). Therefore:
   $\omega = \bigvee_{s \in G_m(\gamma(X))} down_{s'}(m) = \bigvee_{s \in G_m(\gamma(X))} reV_{n,d}(down_s(m)) \preceq reV_{n,d}(\bigvee_{s \in G_m(\gamma(X))} down_s(m))$
   (the last step by lemma 3). We know that $\bigvee_{s \in G_m(\gamma(X))} down_s(m) \in \alpha(\gamma(X))$, and by the Galois connection property, there is $\omega' \in X$ s.t. $\bigvee_{s \in G_m(\gamma(X))} down_s(m) \preceq \omega' \implies \omega \preceq reV_{n,d}(\bigvee_{s \in G_m(\gamma(X))} down_s(m)) \preceq reV_{n,d}(\omega') \in f^\#(X)$. So finally we showed: $\alpha(f(\gamma(X))) \sqsubseteq f^\#(X)$. $\square$

4 Summary

This paper introduced the principles and the theoretical part of our way to analyze programs which use BSTs. Now, with the states and transformers that were defined, we can apply the standard CFG algorithm:

For a given program, create its CFG, initialize the CFG-nodes with the empty abstract state, and begin to update those states using to the abstract transformers. When a fix-point has been reached, or if there were enough iterations, stop updating and check the validity of the abstract state.

Note that the transformers are **sound**, and not necessarily the **best transformers**, and therefore the whole analyzing is an overapproximation. Indeed, one can think of some optimizations for the definitions above, so that the programs' commands would be treated more carefully.