Chapter 4

Trees

4.1 Introduction

The first module of the formalization (TreeM) contains a description of the bare structure of minimalist trees and subtrees of minimalist trees. Furthermore, it contains some functions on trees and nodes.\(^1\)

In the Minimalist Program it is assumed that trees are binary branching, which implies that a node has at most two daughters. Furthermore it is assumed that the nodes of a tree contain features (see also Chapter 5). However, there are four exceptional kinds of nodes that do not seem to contain any features. These four types of empty nodes are described in the following paragraphs.\(^2\)

The first kind of empty node is radically empty. It contains a completely empty feature structure. For instance, intransitive verbs do not select complements. Therefore the label belonging to the sister of the verb is an empty feature structure and consequently the sister of the verb is an empty node. Because of the possibility that nodes are empty, trees can always be binary branching. If a head does not select a complement, the mother of the head still has two daughters, although one of the daughters is an empty node.

A second kind of empty node occurs when a head does not have any phonological content. For example, in the DP girls as opposed to the DP the girls the head of the DP is empty. This is a kind of emptiness that is different from radical emptiness because here the feature structure connected with the head is not empty. It contains, for instance, a category feature.

A third kind of empty node is the one that is created by Merge and

\(^1\)See also [HU79, BGMV93, BRV89].

\(^2\)Note that PRO [Cho81, CL93] and pro [Cho82] are left out of consideration since they are of no importance for the fragment formalized here.
Move as described in Section 1.6. This type does not occur in the representationally oriented formalization described in this work because it only describes the result of the operations Merge and Move, and not how those operations are performed. The stage where the empty node is created is an intermediate stage. Therefore the empty node is not visible in the final results of Merge and Move.

A fourth kind of empty nodes is a trace. Traces are left behind in positions from which a constituent moved. In the formalization it is assumed that traces are in fact copies of the moved constituent.\(^3\) This implies that a movement within a tree is represented by two identical copies of the same constituent in different positions. Furthermore, we will see that there is a link between the positions where the two copies occur.

It is essential for the formalization that copies instead of traces are applied to express movement, since we want to specify the characteristics of the final result of a derivation. In the representational approach chosen here there must be a possibility to dispose over the feature structure of a lexical constituent in every position of a chain, as feature checking applies locally within functional projections. Feature checking is the matching of the features of a moving lexical constituent with the features of the functional projection where it lands. Since traces just are empty nodes without any features they are not appropriate for the local operation of feature checking. Copies, on the other hand, are full-fledged constituents, consisting of nodes labelled with features.

### 4.2 The formalization

**Complete trees** In Definition 4.1, we see the definition of complete trees (\texttt{ComplTrees}). Later on in this chapter I will give the definition of subtrees (\texttt{TreeS}).

**Definition 4.1**

\begin{verbatim}
IMPORT FeatureM

DECL fstruct : FeatureStructS

SORT DirectionS

SORT AddressS == ListS[DirectionS]

OBJ Left : DirectionS
OBJ Right : DirectionS
\end{verbatim}

\(^3\text{Cf. [Cho93, Page 34-35].}\)
4.2. THE FORMALIZATION

AXIOM Elems[DirectionS] = Set(Left,Right)

DECL dir : DirectionS
DECL addr : AddressS

SORT ComplTreeS

FUNC Addresses : ComplTreeS -> FiniteSetS[AddressS]
FUNC Features : ComplTreeS, AddressS -> PARTIAL FeatureStructS
FUNC ConnectionTarget : ComplTreeS, AddressS -> PARTIAL AddressS

DECL ctr : ComplTreeS

AXIOM FORALL addr
(Arrresses ctr1 = Addresses ctr2
And Features(ctr1, addr) = Features(ctr2, addr)
And ConnectionTarget(ctr1, addr) = ConnectionTarget(ctr2, addr))
=> ctr1 = ctr2

In the first line of Definition 4.1, the module \texttt{Features} (see Chapter 5) is imported. Therefore, the variable declaration of \texttt{fstruct} (next line) can be given without defining the sort \texttt{FeatureStructS}.

Next, a definition of addresses to localize nodes in trees is given. Addresses (\texttt{AddressS}) are defined as lists of directions (ListS[DirectionS]). Directions are of a sort (\texttt{DirectionS}) for which two objects are defined: \texttt{Left} and \texttt{Right}. These two directions are the only objects in the set of directions:

AXIOM Elems[DirectionS] = Set(Left,Right)

The reason that there are only two directions lies in the fact that trees are binary branching in the Minimalist Program. In Example 4.1 the address of B is [Left] and the address of G is [Left,Right,Right].

Example 4.1

![Example tree diagram]

To define complete trees we need three characteristic functions: \texttt{Addresses}, \texttt{Features} and \texttt{ConnectionTarget} (see Definition 4.1). These
functions are characteristic for complete trees in the sense that, if two complete trees $ctr_1$ and $ctr_2$ (which are of the sort $\text{CompTreeS}$) yield equal values for all three functions, we can say that $ctr_1$ and $ctr_2$ are equal.

The function $\text{Addresses}$ takes a complete tree and yields a finite set of addresses ($\text{FiniteSetS[AddressS]}$). The set of addresses must be finite because trees are finite in the Minimalist Program and therefore trees have a finite number of nodes (which implies a finite set of addresses of nodes).

The function $\text{Features}$ takes a complete tree and an address and yields a feature structure, which is the feature structure associated with the node that is indicated by the given address. The function is partial ($\text{PARTIAL[FeatureStructS]}$) because it does not always yield a value: it is impossible to name the features of a node at an address which does not exist in the given tree, and furthermore not all nodes contain features, as we will see further on in this chapter.

The function $\text{ConnectionTarget}$ expresses the existence of connections between nodes without a mother-daughter relation within trees. This function is required to represent chains within derivation trees, that is, a node $X$ is the connection target of a node $Y$ if and only if $Y$ is a trace of $X$. Therefore the (complete) trees described in this module can be considered to be directed graphs. They are specified in such a way that they do not only contain arrows from mothers to daughters; it is also possible that two nodes which do not have a mother-daughter relation are linked with an arrow. The connection target of a node with a given address in a given tree is a node with another address in the same tree. Hence, the first address in the function definition of the function $\text{ConnectionTarget}$ indicates the position where the movement starts, and the second address indicates the position that the movement targets. I assume that movement is always leftward (see Section 1.4) so that the first address always is to the right of the second address. The function $\text{ConnectionTarget}$ is partial since not all addresses in a tree have a connection to another node.

Example 4.2

```
    A
   /\  \\
  B   C
 /\  /\  \\
D  E F  G
  \  \     \\
   \   H   I
```

Specifying trees as a kind of directed graphs has the advantage that we do not need indexes to indicate traces. A connection between two nodes I and F
is indicated in the root of the subtree (C) which is the smallest subtree which contains both I and F (Hence $\text{ConnectionTarget}(\text{ctr}, \text{[Right,Right]}) = \text{[Left]}$, where $\text{ctr} = C$, see Example 4.2).

Note that if we want minimalist trees to be real directed graphs we need to define a mother-of relation instead of a mother-of function plus the function $\text{ConnectionTarget}$. The correct way to represent the tree in Example 4.2 would then be:

![Tree Diagram]

**Cutting subtrees** As I claimed above, two complete trees are equal if they are equal with respect to addresses, features and connections. This is what is expressed in the last axiom of Definition 4.1.

For reasons that will become clear in Chapter 8, we need a function which isolates a subtree from a complete tree. The function declaration of this function ($\text{Cut}$) in Definition 4.2 shows that if we indicate an address in a given complete tree, then the function may yield a second complete tree. This second complete tree is a subtree of the original tree with the node with the given address as its root.

Note that subtrees are not always of the sort $\text{TreeS}$ which will be discussed later on in this section. A subtree is of the sort $\text{ComplTreeS}$ if it satisfies all the constraints on $\text{ComplTreeS}$ concerning features, addresses and connections.

**Definition 4.2**

```
PUBLIC Cut : ComplTreeS, AddressS -> PARTIAL ComplTreeS
AXIOM Cut (ctr1, addr1) = ctr2
    <= FORALL addr2, addr3
        ( addr2 In (Addresses ctr2) = (addr1 ++ addr2) In (Addresses ctr1)
            AND Features (ctr2, addr2) = Features (ctr1, addr1 ++ addr2)
            AND ( ConnectionTarget (ctr2, addr2) = addr3
                  <=> ConnectionTarget (ctr1, addr1 ++ addr2) = addr1 ++ addr3
                )
        )
```
There are some constraints on the addresses, features and connections in the tree (\textit{ctr2}) that is yielded (see axiom in Definition 4.2). The addresses, features and connections have to be equal for the original tree (\textit{ctr1}) and the tree that is the result of the function (\textit{ctr2}) in the sense that for each address \textit{addr2} in \textit{ctr2} we can find a corresponding address in \textit{ctr1}. In the following three paragraphs I will give examples with respect to addresses, features and connections.

If the tree in Example 4.3 is \textit{ctr1} and \textit{addr1} is [Right], then C is the root of \textit{ctr2}, and if \textit{addr2} (= [Right,Left] = H) is an address in \textit{ctr2}, then the corresponding address in \textit{ctr1} is [Right,Right,Left] (= H), which is a concatenation (++) of \textit{addr1} and \textit{addr2}.

\textbf{Example 4.3}

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) {A};
  \node (B) at (-1,-2) {B}
    child {node (D) at (-2,-4) {D}};
  \node (C) at (1,-2) {C}
    child {node (E) at (2,-4) {E}};
  \node (G) at (2,-4) {G};
  \node (F) at (1,-4) {F};
  \node (H) at (0,-4) {H};
  \node (I) at (1,-6) {I};
  \node (F) at (0,-4) {F};
  \node (E) at (2,-4) {E};
  \node (D) at (-2,-4) {D};
  \node (C) at (1,-2) {C};
  \node (B) at (-1,-2) {B};
  \node (A) at (0,0) {A};
  \draw (A) -- (B);
  \draw (A) -- (C);
  \draw (B) -- (D);
  \draw (B) -- (E);
  \draw (C) -- (F);
  \draw (C) -- (G);
  \draw (D) -- (H);
  \draw (E) -- (I);
\end{tikzpicture}
\end{center}

With respect to features, if the tree in Example 4.3 is \textit{ctr1} and \textit{addr1} is [Right], then C is the root of \textit{ctr2}, and if \textit{addr2} (= [Right,Left] = H) is an address in \textit{ctr2} with certain features, then the same features can be found in the corresponding address in \textit{ctr1} ([Right,Right,Left] = H), which is a concatenation (++) of \textit{addr1} and \textit{addr2}.

Regarding connections, if the tree in Example 4.3 is \textit{ctr1} and \textit{addr1} is [Right], then C is the root of \textit{ctr2}, and if there is a connection in \textit{ctr2} between \textit{addr2} (= [Right,Left] = H) and \textit{addr} (= [Left] = F), then the corresponding addresses in \textit{ctr1} are respectively [Right,Right,Left] (= H) and [Right,Left] (=F), which are both concatenations of \textit{addr1} and the original addresses in \textit{ctr2}. It is impossible to cut a subtree \textit{ctr2} out of \textit{ctr1} that has a connection to a node outside \textit{ctr2}. For instance, if there is a connection from F to E, it is impossible to cut C out of A, since the address of E is not expressible from the perspective of C: we only have the empty list and all different combinations of [Left] and [Right] at our disposal. Hence, we have the possibility to refer to addresses that are too low for \textit{ctr2} (for instance, [Left,Left]), but not to refer to addresses that are above the root of \textit{ctr2}.

\textbf{Nodes} Nodes are defined by the characteristic functions \texttt{ComplTree} and \texttt{Address}, that is, a node is a part of a given complete tree and it has
an address that determines its position in the tree. Nodes are defined in Definition 4.3. The first axiom defines that two nodes \( nd_1 \) and \( nd_2 \) are equal if they yield the same values for the function \( \text{ComplTree} \) and \( \text{Address} \). The second axiom defines that if \( addr \) is an address belonging to the complete tree \( ctr \), then there must exist a node \( nd \) of which \( ctr \) is the complete tree and of which \( addr \) is the address.

**Definition 4.3**

\[\text{SORT} \ NodeS\]
\[\text{DECL} \ nd : \ NodeS\]
\[\text{FUNC} \ \text{ComplTree} : \ NodeS \rightarrow \ ComplTreeS\]
\[\text{FUNC} \ \text{Address} : \ NodeS \rightarrow \ AddressS\]
\[\text{AXIOM} \ \text{ComplTree} \ nd_1 = \text{ComplTree} \ nd_2\]
\[\text{And} \ \text{Address} \ nd_1 = \text{Address} \ nd_2\]
\[\Rightarrow \ nd_1 = nd_2\]
\[\text{AXIOM} \ \text{addr In (Addresses ctr)}\]
\[\Leftrightarrow \ \text{EXISTS} \ nd\]
\[\left( \ \text{ComplTree} \ nd = ctr\]
\[\text{And} \ \text{Address} \ nd = addr\]
\[\right)\]

**Trees** The characteristic function of the sort of subtrees \( \text{TreeS} \) is \( \text{Root} \). Two subtrees \( tr_1 \) and \( tr_2 \) are equal if they have the same root (first axiom in Definition 4.4). For every tree \( tr \) there is a root \( nd \) (see second axiom). Note that the constraints on \( \text{TreeS} \) are much less stringent than on \( \text{ComplTreeS} \). For instance, a subtree of the sort \( \text{TreeS} \) does not have to contain both nodes of a connection.

The final line of Definition 4.4 defines that there is no difference between nodes \( \text{NodeS} \) and subtrees \( \text{TreeS} \). Hence, all functions that are defined for nodes are also defined for subtrees, and the other way around. This possibility is applied often in the modules described in the following chapters. In the literature some notions are defined on nodes while others are defined on trees. It is convenient to be able to refer, for instance, to the left daughter of a node in the former case, and to the left daughter tree (meaning the left daughter of the root of the tree) in the latter case.

**Definition 4.4**

\[\text{SORT} \ TreeS\]
\[\text{DECL} \ tr : \ TreeS\]
**CHAPTER 4. TREES**

\[
\text{FUNC} \quad \text{Root : TreeS} \rightarrow \text{NodeS}
\]

\[
\text{AXIOM} \quad \text{Root tr}_1 = \text{Root tr}_2 \\
\implies \text{tr}_1 = \text{tr}_2
\]

\[
\text{AXIOM} \quad \exists \text{tr} \left( \text{Root tr} = \text{nd} \right)
\]

\[
\text{SUBSORT} \quad \text{NodeS} \subseteq \text{TreeS}
\]

**Some functions on nodes and trees**  
The functions in Definition 4.5 are all based on the functions `Address`, `ComplTree`, `ConnectionTarget`, `Cut` and/or `Features` as defined above. Note that additional versions of the functions `ConnectionTarget`, `Features` and `Cut` are defined with different arguments.

The function `LeftDaughter` is defined in the following way: the left daughter of a node `nd1` is `nd2` if and only if the complete tree belonging to `nd2` equals the one belonging to `nd1` (hence, if `nd1` and `nd2` are nodes of the same tree), and the address of `nd2` is equal to the address of `nd1` with the direction `[Left]` added at the end of the list. The definition of `RightDaughter` is equal to the definition of `LeftDaughter` except that the direction `[Right]` is added to the address of `nd1`.

The function `ConnectionTarget` defines that there is a connection from the node `nd1` to `nd2` if and only if the complete tree belonging to `nd2` equals the one belonging to `nd1` and the address of `nd2` is equal to the address that is yielded by the function `ConnectionTarget` when it is applied to the complete tree of `nd1` and the address of `nd1`.\(^4\)

The function `Features` yields the features of a node `nd`. The feature of `nd` are equal to the features that are yielded by the function `Features` applied to the complete tree belonging to `nd` and the address of `nd`.

The function `Cut` is defined in the following way. The result of cutting the subtree `tr1` out of a bigger tree is `tr2`, if and only if the address of `tr2` is an empty list (`Nil`) and the complete tree of `tr2` equals the result of the function `Cut` applied to the complete tree of `tr1` and the address of `tr1` (see also Definition 4.2). The address of `tr2` is an empty list as `tr2` is a root. Note that normally roots are nodes, but we define that there is no difference between nodes and trees. Hence, if we want to cut the subtree

\(^4\)Note that the complete trees belonging to `nd1` and `nd2` by definition need to have the same features. In the following chapters we will see that this is not problematic for feature checking in the formalization. In the Minimalist Program features are deleted when checked. This implies that two elements in the same chain cannot be associated with exactly the same feature structure. In the formalization, features are not deleted when checked but instead a 'list' is introduced, which indicates which features are checked at a given point in a chain.
with the root B (tr1) out of the tree in Example 4.1 repeated here as 4.4, then the result tr2 is the tree in Example 4.5.

**Definition 4.5**

- \( \text{FURC LeftDaughter} : \text{NodeS} \rightarrow \text{PARTIAL NodeS} \)
- \( \text{FURC RightDaughter} : \text{NodeS} \rightarrow \text{PARTIAL NodeS} \)
- \( \text{FURC ConnectionTarget} : \text{NodeS} \rightarrow \text{PARTIAL NodeS} \)
- \( \text{FURC Features} : \text{NodeS} \rightarrow \text{FeatureStructS} \)
- \( \text{FURC Cut} : \text{TreeS} \rightarrow \text{TreeS} \)

- **AXIOM** LeftDaughter nd1 = nd2
  
  \[ \iff \text{ComplTree nd2} = \text{ComplTree nd1} \]
  
  And Address nd2 = (Address nd1) ;; Left

- **AXIOM** RightDaughter nd1 = nd2
  
  \[ \iff \text{ComplTree nd2} = \text{ComplTree nd1} \]
  
  And Address nd2 = (Address nd1) ;; Right

- **AXIOM** ConnectionTarget nd1 = nd2
  
  \[ \iff \text{ComplTree nd2} = \text{ComplTree nd1} \]
  
  And Address nd2 = ConnectionTarget (ComplTree nd1, Address nd1)

- **AXIOM** Features nd = Features (ComplTree nd, Address nd)

- **AXIOM** Cut tr1 = tr2
  
  \[ \iff \text{Address tr2} = \text{Nil} \]
  
  And ComplTree tr2 = Cut (ComplTree tr1, Address tr1)

- \( \text{SUBSORT NodeS} \ll \ll \text{FeatureStructS} \)

**Example 4.4**

```
    A
   / \   
  B   C
 / \   
D   E
```

**Example 4.5**

```
    B
   / \   
  D   E
```

```
   / \   
  F   G
```
The function \textbf{Mother} in Definition 4.6 shows that the mother of a node \texttt{nd1} is \texttt{nd2} if and only if \texttt{nd1} is one of the daughters of \texttt{nd2}.

The function \textbf{Sister} in Definition 4.6 shows that the sister of \texttt{nd1} is \texttt{nd2} if and only if \texttt{nd1} has a mother, \texttt{nd1} is not the same node as \texttt{nd2}, and \texttt{nd1} and \texttt{nd2} have the same mother.

A node \texttt{nd1} properly dominates \texttt{nd2} if and only if \texttt{nd1} is the mother of \texttt{nd2}, or if there is a node \texttt{nd3} of which \texttt{nd1} is the mother and \texttt{nd3} properly dominates \texttt{nd2}. Hence, \texttt{B} in Example 4.2 properly dominates \texttt{G} because there is a node \texttt{E} of which \texttt{B} is the mother and \texttt{E} properly dominates \texttt{G} (because \texttt{E} is the mother of \texttt{G}).

A node \texttt{nd1} reflexively dominates \texttt{nd2} if and only if \texttt{nd1} properly dominates \texttt{nd2} or \texttt{nd1} is the same node as \texttt{nd2}. Hence, a node can reflexively dominate itself, but it cannot properly dominate itself.

The node \texttt{nd} is a node of \texttt{(NodeOf)} the subtree \texttt{tr} if and only if the root of \texttt{tr} reflexively dominates \texttt{nd}.

The function \textbf{ConnectionSource} is based on the function \textbf{ConnectionTarget} specified above. A subtree \texttt{tr1} is the starting point (connection source) of a movement to \texttt{tr2} if and only if \texttt{tr2} is the target (connection target) of a movement from \texttt{tr1}.

The function \textbf{IsLeaf} defines that a node \texttt{nd} is a leaf if and only if it does not have any daughters.

The function \textbf{EmptyLeaf} defines that a node \texttt{nd} is an empty leaf if and only if it is a leaf and it does not have a feature structure as a label.

The function \textbf{NonEmptyLeaf} defines that a node \texttt{nd} is an empty leaf if and only if it is a leaf and it does have a feature structure as a label.

\textbf{Definition 4.6}

\begin{verbatim}
FUNC Mother    : NodeS   -> PARTIAL\textbackslash-NodeS
FUNC Sister    : NodeS   -> PARTIAL\textbackslash-NodeS
FUNC PropDominates : NodeS, NodeS -> BoolS
FUNC ReflDominates : NodeS, NodeS -> BoolS
FUNC NodeOf    : NodeS, TreeS  -> BoolS
FUNC ConnectionSource : TreeS  -> PARTIAL\textbackslash-TreeS
FUNC IsLeaf    : NodeS   -> BoolS
FUNC EmptyLeaf : NodeS   -> BoolS
FUNC NonEmptyLeaf : NodeS   -> BoolS

AXIOM Mother nd1 = nd2
  <= LeftDaughter nd2 = nd1
  Or RightDaughter nd2 = nd1

AXIOM Sister nd1 = nd2
  <= Mother nd1 /= Undef
  And nd1 /= nd2
  And Mother nd1 = Mother nd2
\end{verbatim}
4.3. SUMMARY

In this chapter I discussed notions like tree, subtree and node, plus several functions on nodes and trees. Remarkable is that there are two different types of subtrees: those of the sort \textit{TreeS} and those of the sort \textit{ComplTreeS}.

For both subtrees of the former and of the latter type holds that two subtrees are the same if their nodes have the same features and addresses. However, for a subtree \textit{ctr} of the sort \textit{ComplTreeS} there is an additional constraint: if one of the nodes \textit{nd1} of \textit{ctr} is connected with another node \textit{nd2} then \textit{nd2} must also be a node of \textit{ctr}.

The function \textit{ConnectionTarget} represents connections between nodes. I consider minimalist trees to be a kind of directed graphs. The function \textit{ConnectionTarget} is used instead of indices to indicate movements within trees.

The function \textit{Cut} which cuts subtrees out of trees, always yields subtrees of the sort \textit{ComplTreeS}. In Chapter 8 we need the function \textit{Cut} to be able to indicate that the constituents that are linked with each other must be exact copies. When the function \textit{Cut} is applied for this objective, it is essential that the subtrees that are yielded are subtrees with completed connections.