An Isabelle/HOL Formalization of the Textbook Proof of Huffman’s Algorithm

Jasmin Christian Blanchette
Institut für Informatik, Technische Universität München, Germany
blanchette@in.tum.de

May 28, 2015

Abstract

Huffman’s algorithm is a procedure for constructing a binary tree with minimum weighted path length. This report presents a formal proof of the correctness of Huffman’s algorithm written using Isabelle/HOL. Our proof closely follows the sketches found in standard algorithms textbooks, uncovering a few snags in the process. Another distinguishing feature of our formalization is the use of custom induction rules to help Isabelle’s automatic tactics, leading to very short proofs for most of the lemmas.

Contents

1 Introduction 1
  1.1 Binary Codes .................................................. 1
  1.2 Binary Trees ................................................... 2
  1.3 Huffman’s Algorithm ........................................... 4
  1.4 The Textbook Proof ........................................... 5
  1.5 Overview of the Formalization ............................... 6
  1.6 Overview of Isabelle’s HOL Logic ........................... 7
  1.7 Head of the Theory File ...................................... 7

2 Definition of Prefix Code Trees and Forests 8
  2.1 Tree Datatype .................................................. 8
  2.2 Forest Datatype ................................................ 8
  2.3 Alphabet ....................................................... 8

∗This work was supported by the DFG grant NI 491/11-1.
1 Introduction

1.1 Binary Codes

Suppose we want to encode strings over a finite source alphabet to sequences of bits. The approach used by ASCII and most other charsets is to map each source symbol to a distinct $k$-bit code word, where $k$ is fixed and is typically 8 or 16. To encode a string of symbols, we simply encode each symbol in turn. Decoding involves mapping each $k$-bit block back to the symbol it represents.
Fixed-length codes are simple and fast, but they generally waste space. If we know the frequency \( w_a \) of each source symbol \( a \), we can save space by using shorter code words for the most frequent symbols. We say that a (variable-length) code is *optimum* if it minimizes the sum \( \sum_a w_a \delta_a \), where \( \delta_a \) is the length of the binary code word for \( a \). Information theory tells us that a code is optimum if for each source symbol \( c \) the code word representing \( c \) has length
\[
\delta_c = \log_2 \frac{1}{p_c}, \quad \text{where} \quad p_c = \frac{w_c}{\sum_a w_a}.
\]
This number is generally not an integer, so we cannot use it directly. Nonetheless, the above criterion is a useful yardstick and paves the way for arithmetic coding [13], a generalization of the method presented here.

As an example, consider the source string ‘abacabad’. We have
\[
p_a = \frac{1}{2}, \quad p_b = \frac{1}{4}, \quad p_c = \frac{1}{8}, \quad p_d = \frac{1}{8}.
\]
The optimum lengths for the binary code words are all integers, namely
\[
\delta_a = 1, \quad \delta_b = 2, \quad \delta_c = 3, \quad \delta_d = 3,
\]
and they are realized by the code
\[
C_1 = \{a \mapsto 0, b \mapsto 10, c \mapsto 110, d \mapsto 111\}.
\]
Encoding ‘abacabad’ produces the 14-bit code word 01001100100111. The code \( C_1 \) is optimum: No code that unambiguously encodes source symbols one at a time could do better than \( C_1 \) on the input ‘abacabad’. In particular, with a fixed-length code such as
\[
C_2 = \{a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11\}
\]
we need at least 16 bits to encode ‘abacabad’.

1.2 Binary Trees

Inside a program, binary codes can be represented by binary trees. For example, the trees

```
  0
 / \ 1
a   b
 0   1
 0 / \ 1
c   d
```
and

```
  0
 / \ 1
a   b
 0   1
 0 / \ 1
a   b   c   d
```
correspond to $C_1$ and $C_2$. The code word for a given symbol can be obtained as follows: Start at the root and descend toward the leaf node associated with the symbol one node at a time; generate a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The generated sequence of 0s and 1s is the code word.

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no code word is a prefix of another, thereby eliminating the source of all ambiguities. Codes that have this property are called prefix codes. As an example of a code that doesn’t have this property, consider the code associated with the tree

![Tree Diagram]

and observe that ‘bbb’, ‘bd’, and ‘db’ all map to the code word 111.

Each node in a code tree is assigned a weight. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. Code trees can be annotated with their weights:

![Weighted Tree Diagram]

For our purposes, it is sufficient to consider only full binary trees (trees whose inner nodes all have two children). This is because any inner node with only one

---

1Strictly speaking, there is another potential source of ambiguity. If the alphabet consists of a single symbol $a$, that symbol could be mapped to the empty code word, and then any string $aa \ldots a$ would map to the empty bit sequence, giving the decoder no way to recover the original string’s length. This scenario can be ruled out by requiring that the alphabet has cardinality 2 or more.
child can advantageously be eliminated; for example,

\begin{center}
\includegraphics[width=0.5\textwidth]{huffman_tree.png}
\end{center}

becomes

1.3 Huffman’s Algorithm

David Huffman [7] discovered a simple algorithm for constructing an optimum code tree for specified symbol frequencies: Create a forest consisting of only leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as initial weights for the nodes. Then pick the two trees

\begin{center}
\includegraphics[width=0.2\textwidth]{huffman_tree1.png}
\end{center}

and

\begin{center}
\includegraphics[width=0.2\textwidth]{huffman_tree2.png}
\end{center}

with the lowest weights and replace them with the tree

\begin{center}
\includegraphics[width=0.3\textwidth]{huffman_tree3.png}
\end{center}

Repeat this process until only one tree is left.

As an illustration, executing the algorithm for the frequencies

\[ f_d = 3, \quad f_e = 11, \quad f_i = 5, \quad f_s = 7, \quad f_z = 2 \]

gives rise to the following sequence of states:

\begin{align*}
(1) & \quad z & d & f & s & e \\
(2) & \quad 5 & f & s & e \\
\end{align*}
Tree (5) is an optimum tree for the given frequencies.

1.4 The Textbook Proof

Why does the algorithm work? In his article, Huffman gave some motivation but no real proof. For a proof sketch, we turn to Donald Knuth [8, p. 403–404]:

It is not hard to prove that this method does in fact minimize the weighted path length [i.e., $\sum a \delta(a)$], by induction on $m$. Suppose we have $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_m$, where $m \geq 2$, and suppose that we are given a tree that minimizes the weighted path length. (Such a tree certainly exists, since only finitely many binary trees with $m$ terminal nodes are possible.) Let $V$ be an internal node of maximum distance from the root. If $w_1$ and $w_2$ are not the weights already attached to the children of $V$, we can interchange them with the values that are already there; such an interchange does not increase the weighted path length. Thus there is a tree that minimizes the weighted path length and contains the subtree

Now it is easy to prove that the weighted path length of such a tree is minimized if and only if the tree with

```
  w_1
 /   \
|     |
w_2
```

replaced by

```
  w_1 + w_2
```

has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$. 

6
There is, however, a small oddity in this proof: It is not clear why we must assert
the existence of an optimum tree that contains the subtree

Indeed, the formalization works without it.

Cormen et al. [4, p. 385–391] provide a very similar proof, articulated around
the following propositions:

Lemma 16.2
Let \( C \) be an alphabet in which each character \( c \in C \) has frequency \( f[c] \). Let \( x \) and \( y \) be two characters in \( C \) having the lowest frequencies. Then there exists an optimal prefix code for \( C \) in which the codewords for \( x \) and \( y \) have the same length and differ only in the last bit.

Lemma 16.3
Let \( C \) be a given alphabet with frequency \( f[c] \) defined for each character \( c \in C \). Let \( x \) and \( y \) be two characters in \( C \) with minimum frequency. Let \( C' \) be the alphabet \( C \) with characters \( x, y \) removed and (new) character \( z \) added, so that \( C' = C - \{x, y\} \cup \{z\} \); define \( f \) for \( C' \) as for \( C \), except that \( f[z] = f[x] + f[y] \). Let \( T' \) be any tree representing an optimal prefix code for the alphabet \( C' \). Then the tree \( T \), obtained from \( T' \) by replacing the leaf node for \( z \) with an internal node having \( x \) and \( y \) as children, represents an optimal prefix code for the alphabet \( C \).

Theorem 16.4
Procedure HUFFMAN produces an optimal prefix code.

1.5 Overview of the Formalization

This report presents a formalization of the proof of Huffman’s algorithm written using Isabelle/HOL [12]. Our proof is based on the informal proofs given by Knuth and Cormen et al. The development was done independently of Laurent Théry’s Coq proof [14, 15], which through its “cover” concept represents a considerable departure from the textbook proof.

The development consists of 90 lemmas and 5 theorems. Most of them have very short proofs thanks to the extensive use of simplification rules and custom induction rules. The remaining proofs are written using the structured proof format Isar [16] and are accompanied by informal arguments and diagrams.

The report is organized as follows. Section 2 defines the datatypes for binary code trees and forests and develops a small library of related functions. (Incidentally, there is nothing special about binary codes and binary trees. Huffman’s
algorithm and its proof can be generalized to n-ary trees [8, p. 405 and 595].) Section 3 presents a functional implementation of the algorithm. Section 4 defines several tree manipulation functions needed for the proof. Section 5 presents three key lemmas and concludes with the optimality theorem. Section 6 compares our work with Théry’s Coq proof. Finally, Section 7 concludes the report.

1.6 Overview of Isabelle’s HOL Logic

This section presents a brief overview of the Isabelle/HOL logic, so that readers not familiar with the system can at least understand the lemmas and theorems, if not the proofs. Readers who already know Isabelle are encouraged to skip this section.

Isabelle is a generic theorem prover whose built-in metalogic is an intuitionistic fragment of higher-order logic [5, 12]. The metalogical operators are material implication, written \[ \varphi_1; \ldots; \varphi_n \implies \psi \] (“if \( \varphi_1 \) and \ldots and \( \varphi_n \), then \( \psi \)”), universal quantification, written \( \forall x_1 \ldots x_n. \psi \) (“for all \( x_1, \ldots, x_n \) we have \( \psi \)”), and equality, written \( t \equiv u \).

The incarnation of Isabelle that we use in this development, Isabelle/HOL, provides a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers (\( \neg, \land, \lor, \rightarrow \), and \( \exists \)) on terms of type \text{bool}. In addition, = expresses equivalence. The formulas \( \forall x_1 \ldots x_m. \left[ \varphi_1; \ldots; \varphi_n \right] \implies \psi \) and \( \forall x_1, \ldots, x_m. \varphi_1 \land \cdots \land \varphi_n \rightarrow \psi \) are logically equivalent, but they interact differently with Isabelle’s proof tactics.

The term language consists of simply typed \( \lambda \)-terms written in an ML-like syntax [11]. Function application expects no parentheses around the argument list and no commas between the arguments, as in \( f \ x \ y \). Syntactic sugar provides an infix syntax for common operators, such as \( x = y \) and \( x + y \). Types are inferred automatically in most cases, but they can always be supplied using an annotation \( t :: \tau \), where \( t \) is a term and \( \tau \) is its type. The type of total functions from \( \alpha \) to \( \beta \) is written \( \alpha \rightarrow \beta \). Variables may range over functions.

The type of natural numbers is called \text{nat}. The type of lists over type \( \alpha \), written \( \alpha \text{ list} \), features the empty list \([\] \), the infix constructor \( x :: xs \) (where \( x \) is an element of type \( \alpha \) and \( xs \) is a list over \( \alpha \)), and the conversion function \text{set} from lists to sets. The type of sets over \( \alpha \) is written \( \alpha \text{ set} \). Operations on sets are written using traditional mathematical notation.

1.7 Head of the Theory File

The Isabelle theory starts in the standard way.

\begin{verbatim}
theory Huffman
imports Main
begin

We attach the simp attribute to some predefined lemmas to add them to the de-
\end{verbatim}
fault set of simplification rules.

\textbf{declare} \texttt{Int\_Un\_distrib [simp]}
\texttt{Int\_Un\_distrib2 [simp]}
\texttt{max.absorb1 [simp]}
\texttt{max.absorb2 [simp]}

\section{Definition of Prefix Code Trees and Forests}

\subsection{Tree Datatype}

A \textit{prefix code tree} is a full binary tree in which leaf nodes are of the form \texttt{Leaf w a}, where \texttt{a} is a symbol and \texttt{w} is the frequency associated with \texttt{a}, and inner nodes are of the form \texttt{InnerNode w t\textsubscript{1} t\textsubscript{2}}, where \texttt{t\textsubscript{1}} and \texttt{t\textsubscript{2}} are the left and right subtrees and \texttt{w} caches the sum of the weights of \texttt{t\textsubscript{1}} and \texttt{t\textsubscript{2}}. Prefix code trees are polymorphic on the symbol datatype \texttt{a}.

\texttt{datatype} \texttt{a tree =}
\texttt{ Leaf nat a}
\texttt{ InnerNode nat (a tree) (a tree)}

\subsection{Forest Datatype}

The intermediate steps of Huffman’s algorithm involve a list of prefix code trees, or \textit{prefix code forest}.

\texttt{type\_synonym} \texttt{a forest = a tree list}

\subsection{Alphabet}

The \textit{alphabet} of a code tree is the set of symbols appearing in the tree’s leaf nodes.

\texttt{primrec} alphabet :: a tree ⇒ a set \texttt{where}
\texttt{alphabet (Leaf w a) = \{a\}}
\texttt{alphabet (InnerNode w t\textsubscript{1} t\textsubscript{2}) = alphabet t\textsubscript{1} \cup alphabet t\textsubscript{2}}

For sets and predicates, Isabelle gives us the choice between inductive definitions (\texttt{inductive\_set} and \texttt{inductive}) and recursive functions (\texttt{primrec}, \texttt{fun}, and \texttt{function}). In this development, we consistently favor recursion over induction, for two reasons:

- Recursion gives rise to simplification rules that greatly help automatic proof tactics. In contrast, reasoning about inductively defined sets and predicates involves introduction and elimination rules, which are more clumsy than simplification rules.
• Isabelle’s counterexample generator quickcheck [2], which we used extensively during the top-down development of the proof (together with sorry), has better support for recursive definitions.

The alphabet of a forest is defined as the union of the alphabets of the trees that compose it. Although Isabelle supports overloading for non-overlapping types, we avoid many type inference problems by attaching an ‘$F$’ subscript to the forest generalizations of functions defined on trees.

```markdown
primrec alphabet_F :: a forest ⇒ a set where
alphabet_F [] = \emptyset
alphabet_F (t · ts) = alphabet t ∪ alphabet_F ts
```

Alphabets are central to our proofs, and we need the following basic facts about them.

```markdown
lemma finite_alphabet [simp]:
finite (alphabet t)
by (induct t) auto
```

```markdown
lemma exists_in_alphabet:
∃a. a ∈ alphabet t
by (induct t) auto
```

2.4 Consistency

A tree is consistent if for each inner node the alphabets of the two subtrees are disjoint. Intuitively, this means that every symbol in the alphabet occurs in exactly one leaf node. Consistency is a sufficient condition for $\delta_a$ (the length of the unique code word for $a$) to be defined. Although this wellformedness property isn’t mentioned in algorithms textbooks [1, 4, 8], it is essential and appears as an assumption in many of our lemmas.

```markdown
primrec consistent :: a tree ⇒ bool where
consistent (Leaf w a) = True
consistent (InnerNode w t1 t2) =
  (consistent t1 ∧ consistent t2 ∧ alphabet t1 ∩ alphabet t2 = \emptyset)
```

```markdown
primrec consistent_F :: a forest ⇒ bool where
consistent_F [] = True
consistent_F (t · ts) =
  (consistent t ∧ consistent_F ts ∧ alphabet t ∩ alphabet_F ts = \emptyset)
```

Several of our proofs are by structural induction on consistent trees $t$ and involve one symbol $a$. These proofs typically distinguish the following cases.

BASE CASE: $t = \text{Leaf } w \ b$. 10
**INDUCTION STEP:** $t = \text{InnerNode } w \ t_1 \ t_2$.

**SUBCASE 1:** $a$ belongs to $t_1$ but not to $t_2$.

**SUBCASE 2:** $a$ belongs to $t_2$ but not to $t_1$.

**SUBCASE 3:** $a$ belongs to neither $t_1$ nor $t_2$.

Thanks to the consistency assumption, we can rule out the subcase where $a$ belongs to both subtrees.

Instead of performing the above case distinction manually, we encode it in a custom induction rule. This saves us from writing repetitive proof scripts and helps Isabelle’s automatic proof tactics.

**lemma** tree_induct_consistent [consumes 1, case_names base step\_1 step\_2 step\_3]:

\[
\forall w_1 b a. \ P (\text{Leaf } w_1 b) a;
\forall w t_1 t_2 a.
\]

\[
\left[ \begin{array}{l}
\text{consistent } t_1; \ \text{consistent } t_2; \ \text{alphabet } t_1 \cap \text{alphabet } t_2 = \emptyset; \\
a \in \text{alphabet } t_1; \ a \notin \text{alphabet } t_2; \ P t_1 a; \ P t_2 a \end{array} \right] \implies \\
P (\text{InnerNode } w t_1 t_2) a;
\]

\[
\forall w t_1 t_2 a.
\]

\[
\left[ \begin{array}{l}
\text{consistent } t_1; \ \text{consistent } t_2; \ \text{alphabet } t_1 \cap \text{alphabet } t_2 = \emptyset; \\
a \notin \text{alphabet } t_1; \ a \in \text{alphabet } t_2; \ P t_1 a; \ P t_2 a \end{array} \right] \implies \\
P (\text{InnerNode } w t_1 t_2) a;
\]

\[
\forall w t_1 t_2 a.
\]

\[
\left[ \begin{array}{l}
\text{consistent } t_1; \ \text{consistent } t_2; \ \text{alphabet } t_1 \cap \text{alphabet } t_2 = \emptyset; \\
a \notin \text{alphabet } t_1; \ a \notin \text{alphabet } t_2; \ P t_1 a; \ P t_2 a \end{array} \right] \implies \\
P (\text{InnerNode } w t_1 t_2) a
\]

\[
P t a
\]

The proof relies on the induction_schema and lexicographic_order tactics, which automate the most tedious aspects of deriving induction rules. The alternative would have been to perform a standard structural induction on $t$ and proceed by cases, which is straightforward but long-winded.

**apply** rotate_tac
**apply** induction_schema
  **apply** atomize_elim
  **apply** (case_tac $t$)
  **apply** fastforce
  **apply** fastforce
**by** lexicographic_order

The induction_schema tactic reduces the putative induction rule to simpler proof obligations. Internally, it reuses the machinery that constructs the default induction rules. The resulting proof obligations concern (a) case completeness,
(b) invariant preservation (in our case, tree consistency), and (c) wellfoundedness. For tree_conduct_consistent, the obligations (a) and (b) can be discharged using Isabelle’s simplifier and classical reasoner, whereas (c) requires a single invocation of lexicographic_order, a tactic that was originally designed to prove termination of recursive functions [3, 9, 10].

2.5 Symbol Depths

The depth of a symbol (which we denoted by \( \delta_a \) in Section 1.1) is the length of the path from the root to the leaf node labeled with that symbol, or equivalently the length of the code word for the symbol. Symbols that don’t occur in the tree or that occur at the root of a one-node tree have depth 0. If a symbol occurs in several leaf nodes (which may happen with inconsistent trees), the depth is arbitrarily defined in terms of the leftmost node labeled with that symbol.

\[ \text{primrec depth :: } \alpha \text{ tree } \Rightarrow \alpha \Rightarrow \text{nat} \text{ where} \]
\[ \text{depth } (\text{Leaf } w b) a = 0 \]
\[ \text{depth } (\text{InnerNode } w t_1 t_2) a = \]
\[ \begin{cases} 
\text{depth } t_1 a + 1 & \text{if } a \in \text{alphabet } t_1 \\
\text{depth } t_2 a + 1 & \text{else if } a \in \text{alphabet } t_2 \\
0 & \text{else}
\end{cases} \]

The definition may seem very inefficient from a functional programming point of view, but it does not matter, because unlike Huffman’s algorithm, the depth function is merely a reasoning tool and is never actually executed.

2.6 Height

The height of a tree is the length of the longest path from the root to a leaf node, or equivalently the length of the longest code word. This is readily generalized to forests by taking the maximum of the trees’ heights. Note that a tree has height 0 if and only if it is a leaf node, and that a forest has height 0 if and only if all its trees are leaf nodes.

\[ \text{primrec height :: } \alpha \text{ tree } \Rightarrow \text{nat} \text{ where} \]
\[ \text{height } (\text{Leaf } w a) = 0 \]
\[ \text{height } (\text{InnerNode } w t_1 t_2) = \max (\text{height } t_1) (\text{height } t_2) + 1 \]

\[ \text{primrec height } F :: \alpha \text{ forest } \Rightarrow \text{nat} \text{ where} \]
\[ \text{height}_F [\_] = 0 \]
\[ \text{height}_F (t \cdot ts) = \max (\text{height } t) (\text{height}_F ts) \]

The depth of any symbol in the tree is bounded by the tree’s height, and there exists a symbol with a depth equal to the height.
**Lemma depth_le_height:**
\[
\text{depth } t \ a \ \leq \ \text{height } t
\]
**by** \((\text{induct } t)\) auto

**Lemma exists_at_height:**
consistent \(t\) \(\implies\) \(\exists a \in \text{alphabet } t. \ \text{depth } t \ a = \text{height } t\)

**Proof** (induct \(t\))

case \text{Leaf} \hspace{1em} \text{thus} \hspace{1em} \text{case by simp}

next

case (InnerNode \(w t_1 t_2\))

**Note** \(\text{hyps} = \text{InnerNode}\)

let \(t = \text{InnerNode } w t_1 t_2\)

from \(\text{hyps}\) obtain \(b\) where \(b \in \text{alphabet } t_1\) \(\text{depth } t_1 \ b = \text{height } t_1\) **by** auto

from \(\text{hyps}\) obtain \(c\) where \(c \in \text{alphabet } t_2\) \(\text{depth } t_2 \ c = \text{height } t_2\) **by** auto

let \(a = \text{if height } t_1 \geq \text{height } t_2 \ \text{then } b \ \text{else } c\)

from \(b\ c\) have \(a \in \text{alphabet } t\) \(\text{depth } t \ a = \text{height } t\)

**using** \(\text{consistent } b\) **by** auto

thus \(\exists a \in \text{alphabet } t. \ \text{depth } t \ a = \text{height } t\)

qed

The following elimination rules help Isabelle’s classical prover, notably the \texttt{auto} tactic. They are easy consequences of the inequation \(\text{depth } t \ a \leq \text{height } t\).

**Lemma depth_max_heightE_left [elim!]:**
\[
\begin{align*}
& \left[\text{depth } t_1 \ a = \max (\text{height } t_1) (\text{height } t_2); \\
& \quad \left[\text{depth } t_1 \ a = \text{height } t_1; \ \text{height } t_1 \geq \text{height } t_2\right] \implies P\right] \implies P
\end{align*}
\]
**by** \((\text{cut_tac } t = t_1 \ \text{and } a = a \ \text{in depth_le_height})\) simp

**Lemma depth_max_heightE_right [elim!]:**
\[
\begin{align*}
& \left[\text{depth } t_2 \ a = \max (\text{height } t_1) (\text{height } t_2); \\
& \quad \left[\text{depth } t_2 \ a = \text{height } t_2; \ \text{height } t_2 \geq \text{height } t_1\right] \implies P\right] \implies P
\end{align*}
\]
**by** \((\text{cut_tac } t = t_2 \ \text{and } a = a \ \text{in depth_le_height})\) simp

We also need the following lemma.

**Lemma height_gt_0_alphabet_eq_imp_height_gt_0:**
**assumes** \(\text{height } t > 0\) \(\ \text{consistent } t\) \(\text{alphabet } t = \text{alphabet } u\)

**shows** \(\text{height } u > 0\)

**Proof** (cases \(t\))

case \text{Leaf} \hspace{1em} \text{thesis using assms by simp}

next

case (InnerNode \(w t_1 t_2\))

**Note** \(t = \text{InnerNode}\)
from exists_in_alphabet obtain b where b: b ∈ alphabet t₁.
from exists_in_alphabet obtain c where c: c ∈ alphabet t₂.
from b c have bc: b ≠ c using t ⟨consistent t⟩ by fastforce
show thesis
proof (cases u)
  case Leaf thus thesis using b bc ts using by auto
next
  case InnerNode thus thesis by simp
qed
qed

2.7 Symbol Frequencies

The frequency of a symbol (which we denoted by \( w_a \) in Section 1.1) is the sum of the weights attached to the leaf nodes labeled with that symbol. If the tree is consistent, the sum comprises at most one nonzero term. The frequency is then the weight of the leaf node labeled with the symbol, or 0 if there is no such node. The generalization to forests is straightforward.

primrec freq :: α tree ⇒ α ⇒ nat where
freq (Leaf w a) = (λb. if b = a then w else 0)
freq (InnerNode w t₁ t₂) = (λb. freq t₁ b + freq t₂ b)

primrec freqF :: α forest ⇒ α ⇒ nat where
freqF [] = (λb. 0)
freqF (t · ts) = (λb. freq t b + freqF ts b)

Alphabet and symbol frequencies are intimately related. Simplification rules ensure that sums of the form \( \text{freq } t₁ a + \text{freq } t₂ a \) collapse to a single term when we know which tree \( a \) belongs to.

lemma notin_alphabet_imp_freq_0 [simp]:
a /∈ alphabet t ⇒ freq t a = 0
by (induct t) simp+

lemma notin_alphabetF_imp_freqF_0 [simp]:
a /∈ alphabetF ts ⇒ freqF ts a = 0
by (induct ts) simp+

lemma freq_0_right [simp]:
[alphabet t₁ ∩ alphabet t₂ = Ø; a ∈ alphabet t₁] ⇒ freq t₂ a = 0
by (auto intro: notin_alphabet_imp_freq_0 simp: disjoint_iff_not_equal)

lemma freq_0_left [simp]:
[alphabet t₁ ∩ alphabet t₂ = Ø; a ∈ alphabet t₂] ⇒ freq t₁ a = 0
by (auto simp: disjoint_iff_not_equal)
Two trees are comparable if they have the same alphabet and symbol frequencies. This is an important concept, because it allows us to state not only that the tree constructed by Huffman’s algorithm is optimal but also that it has the expected alphabet and frequencies.

We close this section with a more technical lemma.

**Lemma** \( \text{height}_F \_0 \_\_\text{imp}_\_\text{Leaf}_\_\text{freq}_F \_\_\text{in}_\_\text{set} \):

\[
\text{consistent}_F \ ts; \ \text{height}_F \ ts = 0; \ a \in \text{alphabet}_F \ ts \implies \
\text{Leaf} \ (\text{freq}_F \ ts \ a) \ a \in \text{set} \ ts
\]

**Proof** (induct \( ts \))

- **Case Nil** thus case by simp
- **Next**
  - **Case** (Cons \( t \) ts) show case using Cons
    **Proof** (cases \( t \))
    - **Case** Leaf thus thesis using Cons by clarsimp
    **Next**
    - **Case** InnerNode thus thesis using Cons by clarsimp
  qed
qed

### 2.8 Weight

The weight function returns the weight of a tree. In the InnerNode case, we ignore the weight cached in the node and instead compute the tree’s weight recursively. This makes reasoning simpler because we can then avoid specifying cache correctness as an assumption in our lemmas.

**Primrec** weight :: \( \alpha \) tree \( \Rightarrow \) nat where

- weight (Leaf \( w \) \( a \)) = \( w \)
- weight (InnerNode \( w \) \( t_1 \) \( t_2 \)) = weight \( t_1 \) + weight \( t_2 \)

The weight of a tree is the sum of the frequencies of its symbols.

**Lemma** weight_eq_Sum_freq:

\[
\text{consistent} \ t \implies \ \text{weight} \ t = \sum_{a \in \text{alphabet}_F} \text{freq} \ t \ a
\]

by (induct \( t \)) (auto simp: setsum.union_disjoint)

The assumption \( \text{consistent} \ t \) is not necessary, but it simplifies the proof by letting us invoke the lemma setsum.union_disjoint:

\[
\text{finite} \ A; \ \text{finite} \ B; \ A \cap B = \emptyset \implies \sum_{x \in A} g \ x + \sum_{x \in B} g \ x = \sum_{x \in A \cup B} g \ x.
\]
2.9 Cost

The cost of a consistent tree, sometimes called the weighted path length, is given by the sum $\sum_{a \in \text{alphabet } t} \text{freq } t a \times \text{depth } t a$ (which we denoted by $\sum_a w_a \delta_a$ in Section 1.1). It obeys a simple recursive law.

**primrec cost :: α tree ⇒ nat where**
\[
\text{cost } (\text{Leaf } w a) = 0 \\
\text{cost } (\text{InnerNode } w t_1 t_2) = \text{weight } t_1 + \text{cost } t_1 + \text{weight } t_2 + \text{cost } t_2
\]

One interpretation of this recursive law is that the cost of a tree is the sum of the weights of its inner nodes [8, p. 405]. (Recall that $\text{weight } (\text{InnerNode } w t_1 t_2) = \text{weight } t_1 + \text{weight } t_2$.) Since the cost of a tree is such a fundamental concept, it seems necessary to prove that the above function definition is correct.

**theorem cost_eq_Sum_freq_mult_depth:**
consistent $t$ $\implies$ cost $t$ $=$ $\sum_{a \in \text{alphabet } t} \text{freq } t a \times \text{depth } t a$

The proof is by structural induction on $t$. If $t$ $=$ $\text{Leaf } w b$, both sides of the equation simplify to 0. This leaves the case $t$ $=$ $\text{InnerNode } w t_1 t_2$. Let $A$, $A_1$, and $A_2$ stand for $\text{alphabet } t$, $\text{alphabet } t_1$, and $\text{alphabet } t_2$, respectively. We have
\[
\begin{align*}
\text{cost } t &= \text{(definition of cost)} \\
&= \text{weight } t_1 + \text{cost } t_1 + \text{weight } t_2 + \text{cost } t_2 \\
&= \text{(induction hypothesis)} \\
&= \text{weight } t_1 + \sum_{a \in A_1} \text{freq } t_1 a \times \text{depth } t_1 a + \\
&\quad \text{weight } t_2 + \sum_{a \in A_2} \text{freq } t_2 a \times \text{depth } t_2 a \\
&= \text{(definition of depth, consistency)} \\
&= \text{weight } t_1 + \sum_{a \in A_1} \text{freq } t_1 a \times (\text{depth } t_1 a - 1) + \\
&\quad \text{weight } t_2 + \sum_{a \in A_2} \text{freq } t_2 a \times (\text{depth } t_2 a - 1) \\
&= \text{(distributivity of } \times \text{ and } \sum \text{ over } -) \\
&= \text{weight } t_1 + \sum_{a \in A_1} \text{freq } t_1 a \times \text{depth } t_1 a - \sum_{a \in A_1} \text{freq } t_1 a + \\
&\quad \text{weight } t_2 + \sum_{a \in A_2} \text{freq } t_2 a \times \text{depth } t_2 a - \sum_{a \in A_2} \text{freq } t_2 a \\
&= \text{(weight_eq_Sum_freq)} \\
&= \sum_{a \in A_1} \text{freq } t_1 a \times \text{depth } t_1 a + \sum_{a \in A_2} \text{freq } t_2 a \times \text{depth } t_2 a \\
&= \text{(definition of freq, consistency)} \\
&= \sum_{a \in A_1} \text{freq } t a \times \text{depth } t a + \sum_{a \in A_2} \text{freq } t a \times \text{depth } t a \\
&= \text{(setsum.union_disjoint, consistency)} \\
&= \sum_{a \in A_1 \cup A_2} \text{freq } t a \times \text{depth } t a \\
&= \sum_{a \in A} \text{freq } t a \times \text{depth } t a.
\]

The structured proof closely follows this argument.
proof (induct t)
  case Leaf thus case by simp
next
  case (InnerNode w t1 t2)
  let t = InnerNode w t1 t2
  let A = alphabet t and A1 = alphabet t1 and A2 = alphabet t2
  note c = (consistent t)
  note hyps = InnerNode
  have d2: \(\forall a. [A_1 \cap A_2 = \emptyset; a \in A_2] \implies \text{depth t a} = \text{depth t2 a} + 1\)
    by auto
  have cost t = weight t1 + cost t1 + weight t2 + cost t2 by simp
  also have \(\ldots = \text{weight t1} + (\sum_{a \in A_1} \text{freq t1 a} \times \text{depth t1 a}) + \text{weight t2} + (\sum_{a \in A_2} \text{freq t2 a} \times \text{depth t2 a})\)
    using hyps by simp
  also have \(\ldots = \text{weight t1} + (\sum_{a \in A_1} \text{freq t1 a} \times (\text{depth t a} - 1)) + \text{weight t2} + (\sum_{a \in A_2} \text{freq t2 a} \times (\text{depth t a} - 1))\)
    using c d2 by simp
  also have \(\ldots = (\sum_{a \in A_1} \text{freq t1 a} \times \text{depth t a}) + (\sum_{a \in A_2} \text{freq t2 a} \times \text{depth t a})\)
    using c d2 by (simp add: setsum.distrib)
  also have \(\ldots = (\sum_{a \in A_1} \text{freq t1 a} \times \text{depth t a}) + (\sum_{a \in A_2} \text{freq t2 a} \times \text{depth t a})\)
    using c by (simp add: weight_eq_Sum_freq)
  also have \(\ldots = (\sum_{a \in A_1} \text{freq t a} \times \text{depth t a}) + (\sum_{a \in A_2} \text{freq t a} \times \text{depth t a})\)
    using c by auto
  also have \(\ldots = (\sum_{a \in A_1 \cup A_2} \text{freq t a} \times \text{depth t a})\)
    using c by (simp add: setsum.union_disjoint)
  also have \(\ldots = (\sum_{a \in A} \text{freq t a} \times \text{depth t a})\) by simp
  finally show case .
qed

Finally, it should come as no surprise that trees with height 0 have cost 0.

lemma height_0_imp_cost_0 [simp]:
  \(\text{height t} = 0 \implies \text{cost t} = 0\)
  by (case_tac t) simp+

2.10 Optimality

A tree is optimum if and only if its cost is not greater than that of any comparable tree. We can ignore inconsistent trees without loss of generality.
definition optimum :: α tree ⇒ bool where
  optimum t ≡
    ∀ u. consistent u → alphabet t = alphabet u → freq t = freq u →
    cost t ≤ cost u

3 Functional Implementation of Huffman’s Algorithm

3.1 Cached Weight

The cached weight of a node is the weight stored directly in the node. Our arguments rely on the computed weight (embodied by the weight function) rather than the cached weight, but the implementation of Huffman’s algorithm uses the cached weight for performance reasons.

primrec cachedWeight :: α tree ⇒ nat where
  cachedWeight (Leaf w a) = w
  cachedWeight (InnerNode w t1 t2) = w

The cached weight of a leaf node is identical to its computed weight.

lemma height_0_imp_cachedWeight_eq_weight [simp]:
  height t = 0 ⇒ cachedWeight t = weight t
  by (case_tac t) simp

3.2 Tree Union

The implementation of Huffman’s algorithm builds on two additional auxiliary functions. The first one, uniteTrees, takes two trees

\[
\begin{array}{c}
\text{and} \\
\text{and returns the tree} \\
\end{array}
\]

\[
\begin{array}{c}
\text{definition } \textit{uniteTrees} :: \alpha \text{ tree } \Rightarrow \alpha \text{ tree } \Rightarrow \alpha \text{ tree } \text{ where} \\
\textit{uniteTrees} t_1 t_2 ≡ \text{InnerNode (cachedWeight } t_1 + \text{cachedWeight } t_2) \ t_1 \ t_2
\end{array}
\]
The alphabet, consistency, and symbol frequencies of a united tree are easy to connect to the homologous properties of the subtrees.

**Lemma** alphabet_uniteTrees [simp]:
\[
\text{alphabet (uniteTrees } t_1 t_2) = \text{alphabet } t_1 \cup \text{alphabet } t_2
\]
by (simp add: uniteTrees_def)

**Lemma** consistent_uniteTrees [simp]:
\[\text{consistent } t_1; \text{consistent } t_2; \text{alphabet } t_1 \cap \text{alphabet } t_2 = \emptyset \implies \text{consistent (uniteTrees } t_1 t_2)\]
by (simp add: uniteTrees_def)

**Lemma** freq_uniteTrees [simp]:
\[
\text{freq (uniteTrees } t_1 t_2) = (\lambda a. \text{freq } t_1 a + \text{freq } t_2 a)
\]
by (simp add: uniteTrees_def)

### 3.3 Ordered Tree Insertion

The auxiliary function `insortTree` inserts a tree into a forest sorted by cached weight, preserving the sort order.

**Primrec**

\[
\begin{align*}
insortTree &:: 
\alpha \text{ tree } \Rightarrow 
\alpha \text{ forest } \Rightarrow 
\alpha \text{ forest} \\
\text{insortTree } u [ ] &= [ u ] \\
\text{insortTree } u (t \cdot ts) &= \\
&
\quad \text{if cachedWeight } u \leq \text{cachedWeight } t \\
&\quad \text{then } u \cdot t \cdot ts \\
&\quad \text{else } t \cdot \text{insortTree } u ts
\end{align*}
\]

The resulting forest contains one more tree than the original forest. Clearly, it cannot be empty.

**Lemma** length_insortTree [simp]:
\[
\text{length (insortTree } t ts) = \text{length } ts + 1
\]
by (induct ts) simp

**Lemma** insortTree_ne_Nil [simp]:
\[
\text{insortTree } t ts \neq [ ]
\]
by (case_tac ts) simp

The alphabet, consistency, symbol frequencies, and height of a forest after insertion are easy to relate to the homologous properties of the original forest and the inserted tree.

**Lemma** alphabet_F_insortTree [simp]:
\[
\text{alphabet}_F (\text{insortTree } t ts) = \text{alphabet } t \cup \text{alphabet}_F ts
\]
by (induct ts) auto
lemma consistent_F_insortTree [simp]:
consistent_F (insortTree t ts) = consistent_F (t · ts)
by (induct ts) auto

lemma freq_F_insortTree [simp]:
freq_F (insortTree t ts) = (λa. freq t a + freq_F ts a)
by (induct ts) (simp add: ext)+

lemma height_F_insortTree [simp]:
height_F (insortTree t ts) = max (height t) (height_F ts)
by (induct ts) auto

3.4 The Main Algorithm
Huffman’s algorithm repeatedly unites the first two trees of the forest it receives
as argument until a single tree is left. It should initially be invoked with a list of
leaf nodes sorted by weight. Note that it is not defined for the empty list.

fun huffman :: α forest ⇒ α tree where
huffman [t] = t
huffman (t1 · t2 · ts) = huffman (insortTree (uniteTrees t1 t2) ts)

The time complexity of the algorithm is quadratic in the size of the forest. If
we eliminated the inner node’s cached weight component, and instead recom-
puted the weight each time it is needed, the complexity would remain quadratic,
but with a larger constant. Using a binary search in insortTree, the correspond-
ing imperative algorithm is \(O(n \log n)\) if we keep the weight cache and \(O(n^2)\) if
we drop it. An \(O(n)\) imperative implementation is possible by maintaining two
queues, one containing the unprocessed leaf nodes and the other containing the
combined trees [8, p. 404].

The tree returned by the algorithm preserves the alphabet, consistency, and
symbol frequencies of the original forest.

theorem alphabet_huffman [simp]:
ts ≠ [] ⇒ alphabet (huffman ts) = alphabet_F ts
by (induct ts rule: huffman.induct) auto

theorem consistent_huffman [simp]:
[consistent_F ts; ts ≠ []] ⇒ consistent (huffman ts)
by (induct ts rule: huffman.induct) simp+

theorem freq_huffman [simp]:
ts ≠ [] ⇒ freq (huffman ts) = freq_F ts
by (induct ts rule: huffman.induct) (auto simp: ext)
4 Definition of Auxiliary Functions Used in the Proof

4.1 Sibling of a Symbol

The sibling of a symbol $a$ in a tree $t$ is the label of the node that is the (left or right) sibling of the node labeled with $a$ in $t$. If the symbol $a$ is not in $t$'s alphabet or it occurs in a node with no sibling leaf, we define the sibling as being $a$ itself; this gives us the nice property that if $t$ is consistent, then $\text{sibling}_t a \neq a$ if and only if $a$ has a sibling. As an illustration, we have $\text{sibling}_t a = b$, $\text{sibling}_t b = a$, and $\text{sibling}_t c = c$ for the tree

\[
\begin{array}{c}
t = \\
| \\
| a \\
| b \\
| \text{c}
\end{array}
\]

\[
\text{fun sibling :: } \alpha \text{ tree } \Rightarrow \alpha \Rightarrow \alpha \text{ where}
\]

\[
\text{sibling} (\text{Leaf} w b) a = a
\]

\[
\text{sibling} (\text{InnerNode} w (\text{Leaf} w b) (\text{Leaf} w c)) a =
\]

\[
\hspace{1cm} (\text{if } a = b \text{ then } c \text{ else if } a = c \text{ then } b \text{ else } a)
\]

\[
\text{sibling} (\text{InnerNode} w t_1 t_2) a =
\]

\[
\hspace{1cm} (\text{if } a \in \text{alphabet } t_1 \text{ then } \text{sibling}_{t_1} a
\]

\[
\hspace{2cm} \text{else if } a \in \text{alphabet } t_2 \text{ then } \text{sibling}_{t_2} a
\]

\[
\hspace{3cm} \text{else } a)
\]

Because $\text{sibling}$ is defined using sequential pattern matching [9, 10], reasoning about it can become tedious. Simplification rules therefore play an important role.

\[
\text{lemma notin_alphabet_imp_sibling_id [simp]}:
\]

\[
a \notin \text{alphabet } t \Rightarrow \text{sibling}_t a = a
\]

\[
\hspace{1cm} \text{by (cases rule: sibling.cases [where } x = (t, a)])}\]

\[
\text{simp+}
\]

\[
\text{lemma height_0_imp_sibling_id [simp]}:
\]

\[
\text{height } t = 0 \Rightarrow \text{sibling}_t a = a
\]

\[
\hspace{1cm} \text{by (case_tac } t) \text{ simp+}
\]

\[
\text{lemma height_gt_0_in_alphabet_imp_sibling_left [simp]}:
\]

\[
\hspace{1cm} [\text{height } t_1 > 0; a \in \text{alphabet } t_1] \Rightarrow
\]

\[
\text{sibling} (\text{InnerNode } w t_1 t_2) a = \text{sibling}_{t_1} a
\]

\[
\hspace{1cm} \text{by (case_tac } t_1) \text{ simp+}
\]

\[
\text{lemma height_gt_0_in_alphabet_imp_sibling_right [simp]}:
\]

\[
\hspace{1cm} [\text{height } t_2 > 0; a \in \text{alphabet } t_1] \Rightarrow
\]

\[
\text{simp+}
\]
sibling (InnerNode w t₁ t₂) a = sibling t₁ a
by (case_tac t₂) simp⁺

lemma height_gt_0_notin_alphabet_imp_sibling_left [simp]:
⟨height t₁ > 0; a /∈ alphabet t₁⟩ ⇒
sibling (InnerNode w t₁ t₂) a = sibling t₂ a
by (case_tac t₁) simp⁺

lemma height_gt_0_notin_alphabet_imp_sibling_right [simp]:
⟨height t₂ > 0; a /∈ alphabet t₁⟩ ⇒
sibling (InnerNode w t₁ t₂) a = sibling t₂ a
by (case_tac t₂) simp⁺

lemma either_height_gt_0_imp_sibling [simp]:
height t₁ > 0 ∨ height t₂ > 0 ⇒
sibling (InnerNode w t₁ t₂) a =
(if a ∈ alphabet t₁ then sibling t₁ a else sibling t₂ a)
by auto

The following rules are also useful for reasoning about siblings and alphabets.

lemma in_alphabet_imp_sibling_in_alphabet:
a ∈ alphabet t ⇒ sibling t a ∈ alphabet t
by (induct t a rule: sibling.induct) auto

lemma sibling_ne_imp_sibling_in_alphabet:
sibling t a ≠ a ⇒ sibling t a ∈ alphabet t
by (metis notin_alphabet_imp_sibling_id in_alphabet_imp_sibling_in_alphabet)

The default induction rule for sibling distinguishes four cases.

BASE CASE: t = Leaf w b.

INDUCTION STEP 1: t = InnerNode w (Leaf w b) (Leaf w c).

INDUCTION STEP 2: t = InnerNode w (InnerNode w₁ t₁₁ t₁₂) t₂.

INDUCTION STEP 3: t = InnerNode w t₁ (InnerNode w₂ t₂₁ t₂₂).

This rule leaves much to be desired. First, the last two cases overlap and can
normally be handled the same way, so they should be combined. Second, the
nested InnerNode constructors in the last two cases reduce readability. Third, un-
der the assumption that t is consistent, we would like to perform the same case
distinction on a as we did for tree_induct_consistent, with the same benefits for
automation. These observations lead us to develop a custom induction rule that
distinguishes the following cases.

BASE CASE: t = Leaf w b.
INDUCTION STEP 1: \( t = \text{InnerNode} \ w \ (\text{Leaf} \ w \ b) \ (\text{Leaf} \ w \ c) \) with \( b \neq c \).

INDUCTION STEP 2: \( t = \text{InnerNode} \ w \ t_1 \ t_2 \) and either \( t_1 \) or \( t_2 \) has nonzero height.

SUBCASE 1: \( a \) belongs to \( t_1 \) but not to \( t_2 \).

SUBCASE 2: \( a \) belongs to \( t_2 \) but not to \( t_1 \).

SUBCASE 3: \( a \) belongs to neither \( t_1 \) nor \( t_2 \).

The statement of the rule and its proof are similar to what we did for consistent trees, the main difference being that we now have two induction steps instead of one.

**Lemma**: `sibling_induct_consistent`

```
[consumes 1, case_names base step1 step21 step22 step23]:
\[\forall w b a. P (\text{Leaf} \ w \ b) \ a;\]
\[\forall w w_c b a. b \neq c \implies P (\text{InnerNode} \ w \ (\text{Leaf} \ w \ b) \ (\text{Leaf} \ w \ c)) \ a;\]
\[\forall w t_1 t_2 a.\]
\[\forall w t_1 t_2 a.\]
\[\forall w t_1 t_2 a.\]
```

apply `rotate_tac`
apply `induction_schema`
apply `atomize_elim`
apply `(case_tac t, simp)`
apply `clarsimp`
apply `(rename_tac a t_1 t_2)`
apply `(case_tac height t_1 = 0 \land height t_2 = 0)`
apply `simp`
apply `(case_tac t_1)`

23
apply (case_tac t2)
apply fastforce
apply simp+
apply (auto intro: in_alphabet_imp_sibling_in_alphabet)[1]
by lexicographic_order

The custom induction rule allows us to prove new properties of sibling with little effort.

lemma sibling_sibling_id [simp]:
consistent t \implies sibling t (sibling t a) = a
by (induct t a rule: sibling_induct_consistent) simp+

lemma sibling_reciprocal:
[consistent t; sibling t a = b] \implies sibling t b = a
by auto

lemma depth_height_imp_sibling_ne:
[consistent t; depth t a = height t; height t > 0; a \in alphabet t] \implies sibling t a \neq a
by (induct t a rule: sibling_induct_consistent) auto

lemma depth_sibling [simp]:
consistent t \implies depth t (sibling t a) = depth t a
by (induct t a rule: sibling_induct_consistent) simp+

4.2 Leaf Interchange

The swapLeaves function takes a tree t together with two symbols a, b and their frequencies \(w_a\), \(w_b\), and returns the tree t in which the leaf nodes labeled with a and b are exchanged. When invoking swapLeaves, we normally pass freq t a and freq t b for \(w_a\) and \(w_b\).

Note that we do not bother updating the cached weight of the ancestor nodes when performing the interchange. The cached weight is used only in the implementation of Huffman's algorithm, which doesn't invoke swapLeaves.

primrec swapLeaves :: a tree \Rightarrow nat \Rightarrow a \Rightarrow nat \Rightarrow a \Rightarrow a tree where
swapLeaves (Leaf w_c c) w_a a w_b b =
    (if c = a then Leaf w_b b else if c = b then Leaf w_a a else Leaf w_c c)
swapLeaves (InnerNode w t1 t2) w_a a w_b b =
    InnerNode w (swapLeaves t1 w_a a w_b b) (swapLeaves t2 w_a a w_b b)

Swapping a symbol \(a\) with itself leaves the tree \(t\) unchanged if \(a\) does not belong to it or if the specified frequencies \(w_a\) and \(w_b\) equal freq \(t\) \(a\).
lemma swapLeaves_id_when_notin_alphabet [simp]:
a \notin \text{alphabet } t \implies \text{swapLeaves } t \ w \ a \ w' a = t
by (induct t) simp

lemma swapLeaves_id [simp]:
consistent t \implies \text{swapLeaves } t \ (\text{freq } t \ a) \ a \ (\text{freq } t \ a) \ a = t
by (induct t a rule: tree_induct_consistent) simp

The alphabet, consistency, symbol depths, height, and symbol frequencies of the
tree \text{swapLeaves } t \ w_a \ a \ w_b \ b can be related to the homologous properties of \( t \).

lemma alphabet_swapLeaves:
alphabet \ (\text{swapLeaves } t \ w_a \ a \ w_b \ b) =
  (\text{if } a \in \text{alphabet } t \text{ then }
   (\text{if } b \in \text{alphabet } t \text{ then } \text{alphabet } t \setminus \{a\} \cup \{b\}
    \text{ else }
    (\text{if } b \in \text{alphabet } t \text{ then } \text{alphabet } t \setminus \{b\} \cup \{a\}
     \text{ else } \text{alphabet } t))
  \text{ else }
  \text{alphabet } t
by (induct t) auto

lemma consistent_swapLeaves [simp]:
consistent t \implies \text{consistent } (\text{swapLeaves } t \ w_a \ a \ w_b \ b)
by (induct t) (auto simp: alphabet_swapLeaves)

lemma depth_swapLeaves_neither [simp]:
[consistent t; c \neq a; c \neq b] \implies \text{depth } (\text{swapLeaves } t \ w_a \ a \ w_b \ b) \ c = \text{depth } t \ c
by (induct t a rule: tree_induct_consistent) (auto simp: alphabet_swapLeaves)

lemma height_swapLeaves [simp]:
height (\text{swapLeaves } t \ w_a \ a \ w_b \ b) = \text{height } t
by (induct t) simp

lemma freq_swapLeaves [simp]:
\quad [consistent t; a \neq b] \implies
\quad \text{freq } (\text{swapLeaves } t \ w_a \ a \ w_b \ b) =
\quad (\lambda c. \text{if } c = a \text{ then } (\text{if } b \in \text{alphabet } t \text{ then } w_a \text{ else } 0
          \text{ else if } c = b \text{ then } (\text{if } a \in \text{alphabet } t \text{ then } w_b \text{ else } 0
          \text{ else freq } t \ c)
apply (rule ext)
apply (induct t)
by auto

For the lemmas concerned with the resulting tree’s weight and cost, we avoid
subtraction on natural numbers by rearranging terms. For example, we write

\quad \text{weight } (\text{swapLeaves } t \ w_a \ a \ w_b \ b) + \text{freq } t \ a = \text{weight } t + w_b

25
rather than the more conventional

\[ \text{weight} (\text{swapLeaves} \ t \ w_a \ a \ w_b \ b) = \text{weight} \ t + w_b - \text{freq} \ t \ a. \]

In Isabelle/HOL, these two equations are not equivalent, because by definition \( m - n = 0 \) if \( n > m \). We could use the second equation and additionally assert that \( \text{freq} \ t \ a \leq \text{weight} \ t \) (an easy consequence of \text{weight_eq_Sum_freq}), and then apply the \text{arith} tactic, but it is much simpler to use the first equation and stay with \text{simp} and \text{auto}. Another option would be to use integers instead of natural numbers.

\text{lemma weight_swapLeaves}: 
\[
\begin{align*}
\text{if} \ a \in \text{alphabet} \ t \ \text{then} \\
\text{if} \ b \in \text{alphabet} \ t \ \text{then} & \quad \text{weight} (\text{swapLeaves} \ t \ w_a \ a \ w_b \ b) + \text{freq} \ t \ a + \text{freq} \ t \ b = \\
& \quad \text{weight} \ t + w_a + w_b \\
\text{else} & \quad \text{weight} (\text{swapLeaves} \ t \ w_a \ a \ w_b \ b) + \text{freq} \ t \ a = \text{weight} \ t + w_b \\
\text{else} & \quad \text{weight} (\text{swapLeaves} \ t \ w_a \ a \ w_b \ b) + \text{freq} \ t \ b = \text{weight} \ t + w_a \\
\text{else} & \quad \text{weight} (\text{swapLeaves} \ t \ w_a \ a \ w_b \ b) = \text{weight} \ t
\end{align*}
\]

\text{proof (induct} \ t \ \text{a rule: tree_induct_consistent)}

\begin{itemize}
\item \text{BASE CASE: } t = \text{Leaf} \ w \ b
\item \text{case base} \ \text{thus} \ \text{case by clarsimp}
\end{itemize}

\text{next}
\begin{itemize}
\item \text{INDUCTION STEP: } t = \text{InnerNode} \ w \ t_1 \ t_2
\item \text{SUBLCASE 1: } a \ \text{belongs to} \ t_1 \ \text{but not to} \ t_2
\item \text{case} (\text{step} \_1 \ w \ t_1 \ t_2 \ a) \ \text{show} \ \text{case}
\item \text{proof cases}
\item \text{assume} \ b \in \text{alphabet} \ t_1
\item \text{moreover hence} \ b \notin \text{alphabet} \ t_2 \ \text{using} \ \text{step} \_1 \ \text{by auto}
\item \text{ultimately show} \ \text{case using} \ \text{step} \_1 \ \text{by simp}
\item \text{next}
\item \text{assume} \ b \notin \text{alphabet} \ t_1 \ \text{thus} \ \text{case using} \ \text{step} \_1 \ \text{by auto}
\item \text{qed}
\item \text{next}
\item \text{SUBLCASE 2: } a \ \text{belongs to} \ t_2 \ \text{but not to} \ t_1
\item \text{case} (\text{step} \_2 \ w \ t_1 \ t_2 \ a) \ \text{show} \ \text{case}
\item \text{proof cases}
\item \text{assume} \ b \in \text{alphabet} \ t_1
\end{itemize}
moreover hence \( b \notin \text{alphabet } t_2 \) using step 2 by auto
ultimately show case using step 2 by simp
next
assume \( b \notin \text{alphabet } t_1 \) thus case using step 2 by auto
qed
next
— SUBCASE 3: \( a \) belongs to neither \( t_1 \) nor \( t_2 \)
case (step 3 \( w t_1 t_2 a \)) show case
proof cases
assume \( b \in \text{alphabet } t_1 \)
moreover hence \( b \notin \text{alphabet } t_2 \) using step 3 by auto
ultimately show case using step 3 by simp
next
assume \( b \notin \text{alphabet } t_1 \) thus case using step 3 by auto
qed
qed

lemma cost_swapLeaves:
\[ \text{(consistent } t; a \neq b) \implies \]
if \( a \in \text{alphabet } t \) then
if \( b \in \text{alphabet } t \) then
\[ \text{cost (swapLeaves } t \ a \ a \ w \ b) + \text{freq } t \ a \times \text{depth } t \ a \]
\[ + \text{freq } t \ b \times \text{depth } t \ b = \]
\[ \text{cost } t + w_a \times \text{depth } t \ b + w_b \times \text{depth } t \ a \]
else
\[ \text{cost (swapLeaves } t \ a \ a \ w \ b) + \text{freq } t \ a \times \text{depth } t \ a = \]
\[ \text{cost } t + w_b \times \text{depth } t \ a \]
else
if \( b \in \text{alphabet } t \) then
\[ \text{cost (swapLeaves } t \ a \ a \ w \ b) + \text{freq } t \ b \times \text{depth } t \ b = \]
\[ \text{cost } t + w_a \times \text{depth } t \ b \]
else
\[ \text{cost (swapLeaves } t \ a \ a \ w \ b) = \text{cost } t \]
proof (induct \( t \))
case Leaf show case by simp
next
case (InnerNode \( w t_1 t_2 \))
note \( c = \langle \text{consistent } (\text{InnerNode } w t_1 t_2) \rangle \)
note \( \text{hyps = InnerNode} \)
have \( \bar{w}_1 \): if \( a \in \text{alphabet } t_1 \) then
if \( b \in \text{alphabet } t_1 \) then
weight (swapLeaves \( t_1 \ a \ a \ w \ b \)) + freq \( t_1 \ a \) + freq \( t_1 \ b \) =
weight \( t_1 + w_a + w_b \)
else
\[
\text{weight} (\text{swapLeaves} \ t_1 \ w_a \ a \ w_b \ b) + \text{freq} \ t_1 \ a = \text{weight} \ t_1 + w_b
\]
else
if \( b \in \text{alphabet} \ t_1 \) then
\[
\text{weight} (\text{swapLeaves} \ t_1 \ w_a \ a \ w_b \ b) + \text{freq} \ t_1 \ b = \text{weight} \ t_1 + w_a
\]
else
\[
\text{weight} (\text{swapLeaves} \ t_1 \ w_a \ a \ w_b \ b) = \text{weight} \ t_1 \ \text{using hyps}
\]
by \((\text{simp add: weight_swapLeaves})\)
\begin{align*}
\text{have } w_2: & \text{ if } a \in \text{alphabet} \ t_2 \ \text{then} \\
& \text{ weight} (\text{swapLeaves} \ t_2 \ w_a \ a \ w_b \ b) + \text{freq} \ t_2 \ a + \text{freq} \ t_2 \ b = \\
& \text{ weight} \ t_2 + w_a + w_b \\
& \text{ else } \\
& \text{ weight} (\text{swapLeaves} \ t_2 \ w_a \ a \ w_b \ b) + \text{freq} \ t_2 \ a = \text{weight} \ t_2 + w_b \\
& \text{ else } \\
& \text{ weight} (\text{swapLeaves} \ t_2 \ w_a \ a \ w_b \ b) + \text{freq} \ t_2 \ b = \text{weight} \ t_2 + w_a \\
& \text{ else } \\
& \text{ weight} (\text{swapLeaves} \ t_2 \ w_a \ a \ w_b \ b) = \text{weight} \ t_2 \ \text{using hyps} \\
\end{align*}
by \((\text{simp add: weight_swapLeaves})\)
\begin{align*}
\text{show } \text{case} \\
\text{proof } \text{cases} \\
& \text{ assume } a_1: a \in \text{alphabet} \ t_1 \\
& \text{ hence } a_2: a \notin \text{alphabet} \ t_2 \ \text{using c} \ \text{by auto} \\
& \text{ show } \text{case} \\
& \text{ proof } \text{cases} \\
& \text{ assume } b_1: b \in \text{alphabet} \ t_1 \\
& \text{ hence } b \notin \text{alphabet} \ t_2 \ \text{using c} \ \text{by auto} \\
& \text{ thus } \text{case using } a_1 \ a_2 \ b_1 \ w_1 \ w_2 \ \text{hyps} \ \text{by simp} \\
& \text{ next} \\
& \text{ assume } b_1: b \notin \text{alphabet} \ t_1 \ \text{show case} \\
& \text{ proof } \text{cases} \\
& \text{ assume } b \in \text{alphabet} \ t_2 \ \text{thus case using } a_1 \ a_2 \ b_1 \ w_1 \ w_2 \ \text{hyps} \ \text{by simp} \\
& \text{ next} \\
& \text{ assume } b \notin \text{alphabet} \ t_2 \ \text{thus case using } a_1 \ a_2 \ b_1 \ w_1 \ w_2 \ \text{hyps} \ \text{by simp} \\
& \text{ qed} \\
& \text{ qed} \\
& \text{ next} \\
& \text{ assume } a_1: a \notin \text{alphabet} \ t_1 \ \text{show case} \\
& \text{ proof } \text{cases} \\
& \text{ assume } a_2: a \in \text{alphabet} \ t_2 \ \text{show case} \\
& \text{ proof } \text{cases} \\
& \text{ assume } b_1: b \in \text{alphabet} \ t_1
\end{align*}
Common sense tells us that the following statement is valid: “If Astrid exchanges her house with Bernard’s neighbor, Bernard becomes Astrid’s new neighbor.” A similar property holds for binary trees.

**Lemma** sibling_swapLeaves_sibling [simp]:

\[
\text{consistent } t; \text{ sibling } t b \neq b; a \neq b \Rightarrow \text{sibling (swapLeaves } t w a w s \text{ (sibling } t b)) a = b
\]

**Proof** (induct t)

- case Leaf thus case simp

**Next**

- case (InnerNode w t1 t2)
- note hyps = InnerNode
- show case
- proof (cases height t1 = 0)
  - case True
  - note h1 = True
show thesis
proof (cases \( t_1 \))
  case (Leaf \( w_c \ c \) )
  note \( l_1 = \text{Leaf} \)
  show thesis
proof (cases height \( t_2 = 0 \))
  case True
  note \( h_2 = \text{True} \)
  show thesis
proof (cases \( t_2 \))
  case Leaf thus thesis using \( l_1 \) hyps by auto metis
next
  case InnerNode thus thesis using \( h_2 \) by simp
qed
next
  case False
  note \( h_2 = \text{False} \)
  show thesis
proof cases
  assume \( c = b \) thus thesis using \( l_1 \) \( h_2 \) hyps by simp
next
  assume \( c \neq b \)
  have sibling \( t_2 \) \( b \in \text{alphabet} \) \( t_2 \) using \( c \neq b \) \( l_1 \) \( h_2 \) hyps
  by (simp add: sibling_ne_imp_sibling_in_alphabet)
  thus thesis using \( c \neq b \) \( l_1 \) \( h_2 \) hyps by auto
qed
next
  case InnerNode thus thesis using \( h_1 \) by simp
qed
next
  case False
  note \( h_1 = \text{False} \)
  show thesis
proof (cases height \( t_2 = 0 \))
  case True
  note \( h_2 = \text{True} \)
  show thesis
proof (cases \( t_2 \))
  case (Leaf \( w_d \ d \) )
  note \( l_2 = \text{Leaf} \)
  show thesis
proof cases

30
assume \( d = b \) thus thesis using \( h_1 \) \( h_2 \) hyps by simp

next

assume \( d \neq b \) show thesis
proof (cases \( b \in \text{alphabet } t_1 \))

case True
hence sibling \( t_1 b \in \text{alphabet } t_1 \) using \( d \neq b \) \( h_1 \) \( h_2 \) hyps
by (simp add: sibling_ne_imp_sibling_in_alphabet)
thus thesis using True \( d \neq b \) \( h_1 \) \( h_2 \) hyps
by (simp add: alphabet_swapLeaves)

next

case False thus thesis using \( d \neq b \) \( h_1 \) \( h_2 \) hyps by simp
qed

next

case InnerNode thus thesis using \( h_2 \) by simp

next

case False
note \( h_2 = \text{False} \)
show thesis
proof (cases \( b \in \text{alphabet } t_1 \))

case True thus thesis using \( h_1 \) \( h_2 \) hyps by auto

next

case False
note \( b_1 = \text{False} \)
show thesis
proof (cases \( b \in \text{alphabet } t_2 \))

case True thus thesis using \( b_1 \) \( h_1 \) \( h_2 \) hyps
by (auto simp: in_alphabet_imp_sibling_in_alphabet
alphabet_swapLeaves)

next

case False thus thesis using \( b_1 \) \( h_1 \) \( h_2 \) hyps by simp
qed

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

4.3 Symbol Interchange

The swapSyms function provides a simpler interface to swapLeaves, with \( freq \ t \ a \) and \( freq \ t \ b \) in place of \( w_a \) and \( w_b \). Most lemmas about swapSyms are directly adapted from the homologous results about swapLeaves.
definition swapSyms :: α tree ⇒ α ⇒ α ⇒ α tree
where
swapSyms t a b ≡ swapLeaves t (freq t a) a (freq t b) b

lemma swapSyms_id [simp]:
consistent t ⇒ swapSyms t a a = t
by (simp add: swapSyms_def)

lemma alphabet_swapSyms [simp]:
[a ∈ alphabet t; b ∈ alphabet t] ⇒ alphabet (swapSyms t a b) = alphabet t
by (simp add: swapSyms_def alphabet_swapLeaves)

lemma consistent_swapSyms [simp]:
consistent t ⇒ consistent (swapSyms t a b)
by (simp add: swapSyms_def)

lemma depth_swapSyms_neither [simp]:
[consistent t; c ≠ a; c ≠ b] ⇒ depth (swapSyms t a b) c = depth t c
by (simp add: swapSyms_def)

lemma freq_swapSyms [simp]:
[consistent t; a ∈ alphabet t; b ∈ alphabet t] ⇒ freq (swapSyms t a b) = freq t
by (case_tac a = b) (simp add: swapSyms_def ext)

lemma cost_swapSyms:
assumes consistent t a ∈ alphabet t b ∈ alphabet t
shows cost (swapSyms t a b) + freq t a × depth t a + freq t b × depth t b =
cost t + freq t a × depth t b + freq t b × depth t a
proof cases
assume a = b thus thesis using assms by simp
next
assume a ≠ b
moreover hence cost (swapLeaves t (freq t a) a (freq t b) b)
+ freq t a × depth t a + freq t b × depth t b =
cost t + freq t a × depth t b + freq t b × depth t a
using assms by (simp add: cost_swapLeaves)
ultimately show thesis using assms by (simp add: swapSyms_def)
qed

If a’s frequency is lower than or equal to b’s, and a is higher up in the tree than b
or at the same level, then interchanging a and b does not increase the tree’s cost.

lemma le_le_imp_sum_mult_le_sum_mult:
[i ≤ j; m ≤ (n::nat)] ⇒ i × n + j × m ≤ i × m + j × n
apply (subgoal_tac i × m + i × (n - m) + j × m ≤ i × m + j × m + j × (n - m))
apply (simp add: diff_mult_distrib2)
by simp

lemma cost_swapSyms_le:
assumes consistent t a ∈ alphabet t b ∈ alphabet t freq t a ≤ freq t b
depth t a ≤ depth t b
shows cost (swapSyms t a b) ≤ cost t
proof –
  let aabb = freq t a × depth t a + freq t b × depth t b
  let abba = freq t a × depth t b + freq t b × depth t a
  have abba ≤ aabb using assms(4−5)
    by (rule le_le_imp_sum_mult_le_sum_mult)
  have cost (swapSyms t a b) + aabb = cost t + abba using assms(1−3)
    by (simp add: cost_swapSyms add.assoc [THEN sym])
  also have . . . ≤ cost t + aabb using abba ≤ aabb by simp
  finally show thesis using assms(4−5) by simp
qed

As stated earlier, “If Astrid exchanges her house with Bernard’s neighbor, Bernard becomes Astrid’s new neighbor.”

lemma sibling_swapSyms_sibling [simp]:
\[\text{consistent t; sibling t b \neq b; a \neq b} \implies\]
sibling (swapSyms t a (sibling t b)) a = b
by (simp add: swapSyms_def)

“If Astrid exchanges her house with Bernard, Astrid becomes Bernard’s old neighbor’s new neighbor.”

lemma sibling_swapSyms_other_sibling [simp]:
\[\text{consistent t; sibling t b \neq a; sibling t b \neq b; a \neq b} \implies\]
sibling (swapSyms t a b) (sibling t b) = a
by (metis consistent_swapSyms sibling_swapSyms_sibling sibling_reciprocal)

4.4 Four-Way Symbol Interchange

The \text{swapSyms} function exchanges two symbols \(a\) and \(b\). We use it to define the four-way symbol interchange function \text{swapFourSyms}, which takes four symbols \(a, b, c, d\) with \(a \neq b\) and \(c \neq d\), and exchanges them so that \(a\) and \(b\) occupy \(c\) and \(d\)’s positions.

A naive definition of this function would be

\[
\text{swapFourSyms t a b c d} \equiv \text{swapSyms (swapSyms t a c) b d}.
\]

This definition fails in the face of aliasing: If \(a = d\), but \(b \neq c\), then \text{swapFourSyms}
a b c d would leave a in b’s position.\(^2\)

**definition** swapFourSyms : α tree ⇒ α ⇒ α ⇒ α ⇒ α tree where

\[
\text{swapFourSyms } t \ a \ b \ c \ d \equiv
\begin{align*}
\text{if } a &= d \text{ then swapSyms } t \ b \ c \\
\text{else if } b &= c \text{ then swapSyms } t \ a \ d \\
\text{else swapSyms (swapSyms } t \ a \ c) \ b \ d
\end{align*}
\]

Lemmas about swapFourSyms are easy to prove by expanding its definition.

**lemma** alphabet_swapFourSyms [simp]:

\[
\text{alphabet (swapFourSyms } t \ a \ b \ c \ d) = \text{alphabet } t
\]

by (simp add: swapFourSyms_def)

**lemma** consistent_swapFourSyms [simp]:

\[
\text{consistent } t \Rightarrow \text{consistent (swapFourSyms } t \ a \ b \ c \ d)
\]

by (simp add: swapFourSyms_def)

**lemma** freq_swapFourSyms [simp]:

\[
\text{freq (swapFourSyms } t \ a \ b \ c \ d) = \text{freq } t
\]

by (auto simp: swapFourSyms_def)

More Astrid and Bernard insanity: “If Astrid and Bernard exchange their houses with Carmen and her neighbor, Astrid and Bernard will now be neighbors.”

**lemma** sibling_swapFourSyms_when_4th_is_sibling:

assumes consistent \( t \ a \in \text{alphabet } t \ b \in \text{alphabet } t \ c \in \text{alphabet } t \ a \neq b \ \text{sibling } t \ c \neq c \)

shows sibling (swapFourSyms \( t \ a \ b \ c \) (sibling \( t \ c \))) \( a = b \)

proof (cases \( a \neq \text{sibling } t \ c \land b \neq c \))

\begin{itemize}
\item(case True show thesis \)
\item(proof –
\begin{itemize}
\item(let \( d = \text{sibling } t \ c \)
\item(let \( t_4 = \text{swapFourSyms } t \ a \ b \ c \ d \)
\item(have abba: (sibling \( t_4 \ a = b ) = (\text{sibling } t_4 b = a\) ) using (consistent \( t \) )
\item(by (metis consistent_swapFourSyms sibling_reciprocal)
\item(have s: sibling \( t \ c = \text{sibling } (\text{swapSyms } t \ a \ c) \ ) a using True assms
\item(by (metis sibling_reciprocal sibling_swapSyms_sibling)
\item(have sibling \( t_4 \ b = \text{sibling } (\text{swapSyms } t \ a \ c) \ ) d using s True assms
\item(by (auto simp: swapFourSyms_def)
\item(also have \( \ldots = \ a \) using True assms
\end{itemize}
\end{itemize}
\end{itemize}

\(^2\)Cormen et al. [4, p. 390] forgot to consider this case in their proof. Thomas Cormen indicated in a personal communication that this will be corrected in the next edition of the book.
by (metis sibling_reciprocal sibling_swapSyms_other_sibling
    swapLeaves_id swapSyms_def)

finally have sibling \( t_s \ b = a \).
with abba show thesis ..
qed

next

\textbf{case False} thus thesis using assms
by (auto intro: sibling_reciprocal simp: swapFourSyms_def)
qed

\subsection{Sibling Merge}

Given a symbol \( a \), the \texttt{mergeSibling} function transforms the tree

\[ \text{Leaf} w b b \]
\[ \text{InnerNode} w (\text{Leaf} w b b) (\text{Leaf} w c c) \]

into

\[ \text{Leaf} w (b + c) a \]
\[ \text{InnerNode} w (\text{Leaf} w b b) (\text{Leaf} w c c) \]

The frequency of \( a \) in the result is the sum of the original frequencies of \( a \) and \( b \), so as not to alter the tree’s weight.

\begin{verbatim}
fun mergeSibling :: \( \alpha \) tree \Rightarrow \( \alpha \) \Rightarrow \( \alpha \) tree where
mergeSibling (Leaf \( w \) \( b \) \( b \)) \( a \) = Leaf \( w \) \( b \) \( b \)
mergeSibling (InnerNode \( w \) (Leaf \( w \) \( b \) \( b \)) (Leaf \( w \) \( c \) \( c \))) \( a \) =
    (if \( a = b \lor a = c \) then Leaf (\( w_b \) \( + \) \( w_c \)) \( a \)
    else InnerNode \( w \) (Leaf \( w_b \) \( b \)) (Leaf \( w_c \) \( c \))
mergeSibling (InnerNode \( w \) \( t_1 \) \( t_2 \)) \( a \) =
    InnerNode \( w \) (mergeSibling \( t_1 \) \( a \)) (mergeSibling \( t_2 \) \( a \))
\end{verbatim}

The definition of \texttt{mergeSibling} has essentially the same structure as that of \texttt{sibling}. As a result, the custom induction rule that we derived for \texttt{sibling} works equally well for reasoning about \texttt{mergeSibling}.

\textbf{lemmas} mergeSibling_induct_consistent = sibling_induct_consistent

The properties of \texttt{mergeSibling} echo those of \texttt{sibling}. Like with \texttt{sibling}, simplification rules are crucial.
lemma notin_alphabet_imp_mergeSibling_id [simp]:
a ∉ alphabet t =⇒ mergeSibling t a = t
by (induct t a rule: mergeSibling.induct) simp+

lemma height_gt_0_imp_mergeSibling_left [simp]:
height t₁ > 0 =⇒
mergeSibling (InnerNode w t₁ t₂) a =
  InnerNode w (mergeSibling t₁ a) (mergeSibling t₂ a)
by (case_tac t₁) simp+

lemma height_gt_0_imp_mergeSibling_right [simp]:
height t₂ > 0 =⇒
mergeSibling (InnerNode w t₁ t₂) a =
  InnerNode w (mergeSibling t₁ a) (mergeSibling t₂ a)
by (case_tac t₂) simp+

lemma either_height_gt_0_imp_mergeSibling [simp]:
height t₁ > 0 ∨ height t₂ > 0 =⇒
mergeSibling (InnerNode w t₁ t₂) a =
  InnerNode w (mergeSibling t₁ a) (mergeSibling t₂ a)
by auto

lemma alphabet_mergeSibling [simp]:
[consistent t; a ∈ alphabet t] =⇒
alphabet (mergeSibling t a) = (alphabet t − {sibling t a}) ∪ {a}
by (induct t a rule: mergeSibling_induct_consistent) auto

lemma consistent_mergeSibling [simp]:
consistent t =⇒ consistent (mergeSibling t a)
by (induct t a rule: mergeSibling_induct_consistent) auto

lemma freq_mergeSibling:
[consistent t; a ∈ alphabet t; sibling t a ≠ a] =⇒
freq (mergeSibling t a) =
  (λc. if c = a then freq t a + freq t (sibling t a)
  else if c = sibling t a then 0
  else freq t c)
by (induct t a rule: mergeSibling_induct_consistent)
  (auto simp: fun_eq_iff)

lemma weight_mergeSibling [simp]:
weight (mergeSibling t a) = weight t
by (induct t a rule: mergeSibling.induct) simp+

If a has a sibling, merging a and its sibling reduces t’s cost by freq t a + freq t (sibling t a).
lemma cost_mergeSibling:
\[ \text{consistent } t, \text{ sibling } t \; a \neq a \implies \text{cost } (\text{mergeSibling } t \; a) + \text{freq } t \; a + \text{freq } t \; (\text{sibling } t \; a) = \text{cost } t \]
by (induct t \; a rule: mergeSibling_induct_consistent) auto

4.6 Leaf Split

The splitLeaf function undoes the merging performed by mergeSibling: Given two symbols \(a, b\) and two frequencies \(w_a, w_b\), it transforms

\[
\text{Leaf } \begin{array}{c} a \\ w_a \end{array} \begin{array}{c} b \\ w_b \end{array}
\]

into

\[
\begin{array}{c} a \\ w_a \end{array} \begin{array}{c} b \\ w_b \end{array}
\]

In the resulting tree, \(a\) has frequency \(w_a\) and \(b\) has frequency \(w_b\). We normally invoke it with \(w_a\) and \(w_b\) such that \(\text{freq } t \; a = w_a + w_b\).

primrec splitLeaf :: \(\alpha\) tree \Rightarrow \(\mathbb{N}\) \Rightarrow \(\alpha\) \Rightarrow \(\alpha\) \Rightarrow \(\alpha\) tree
where
\[
\text{splitLeaf } (\text{Leaf } w \; c) \; w_a \; a \; w_b \; b = \\
(\text{if } c = a \text{ then } \text{InnerNode } w \; c (\text{Leaf } w \; a) \; (\text{Leaf } w \; b) \text{ else } \text{Leaf } w \; c)
\]
\[
\text{splitLeaf } (\text{InnerNode } w \; t_1 \; t_2) \; w_a \; a \; w_b \; b = \\
\text{InnerNode } w (\text{splitLeaf } t_1 \; w_a \; a \; w_b \; b) \; (\text{splitLeaf } t_2 \; w_a \; a \; w_b \; b)
\]

primrec splitLeaf_f :: \(\alpha\) forest \Rightarrow \(\mathbb{N}\) \Rightarrow \(\alpha\) \Rightarrow \(\alpha\) \Rightarrow \(\alpha\) \Rightarrow \(\alpha\) forest
where
\[
\text{splitLeaf_f } [] \; w_a \; a \; w_b \; b = []
\]
\[
\text{splitLeaf_f } (t \cdot ts) \; w_a \; a \; w_b \; b = \\
\text{splitLeaf_f } t \; w_a \; a \; w_b \; b \cdot \text{splitLeaf_f } ts \; w_a \; a \; w_b \; b
\]

Splitting leaf nodes affects the alphabet, consistency, symbol frequencies, weight, and cost in unsurprising ways.

lemma notin_alphabet_imp_splitLeaf_id [simp]:
\[ a \notin \text{alphabet } t \implies \text{splitLeaf } t \; w_a \; a \; w_b \; b = t \]
by (induct t) simp+

lemma notin_alphabet_f_imp_splitLeaf_f_id [simp]:
\[ a \notin \text{alphabet } t \; ts \implies \text{splitLeaf_f } ts \; w_a \; a \; w_b \; b = ts \]
by (induct ts) simp+
lemma alphabet_splitLeaf [simp]:
alphabet (splitLeaf t a w b) = 
  (if a ∈ alphabet t then alphabet t ∪ {b} else alphabet t)
by (induct t) simp+

lemma consistent_splitLeaf [simp]:
[consistent t; b /∈ alphabet t] ⇒ consistent (splitLeaf t a w b)
by (induct t) auto

lemma freq_splitLeaf [simp]:
[consistent t; b /∈ alphabet t] ⇒ freq (splitLeaf t a w b) = 
  (if a ∈ alphabet t then 
      (λc. if c = a then w else if c = b then w else freq t c) 
    else 
      freq t)
by (induct t b rule: tree_induct_consistent) (rule ext, auto)+

lemma weight_splitLeaf [simp]:
[consistent t; a ∈ alphabet t; freq t a = w_a + w_b] ⇒ 
weight (splitLeaf t a w b) = weight t
by (induct t a rule: tree_induct_consistent) simp+

lemma cost_splitLeaf [simp]:
[consistent t; a ∈ alphabet t; freq t a = w_a + w_b] ⇒ 
cost (splitLeaf t a w b) = cost t + w_a + w_b
by (induct t a rule: tree_induct_consistent) simp+

4.7 Weight Sort Order
An invariant of Huffman’s algorithm is that the forest is sorted by weight. This
is expressed by the sortedByWeight function.

fun sortedByWeight :: a forest ⇒ bool where
sortedByWeight [] = True
sortedByWeight [t] = True
sortedByWeight (t1 · t2 · ts) = 
  (weight t1 ≤ weight t2 ∧ sortedByWeight (t2 · ts))

The function obeys the following fairly obvious laws.

lemma sortedByWeight_Cons_imp_sortedByWeight:
sortedByWeight (t · ts) ⇒ sortedByWeight ts
by (case_tac ts) simp+
lemma sortedByWeight_Cons_imp_forall_weight_ge:
sortedByWeight (t · ts) ⟷ ∀u ∈ set ts. weight u ≥ weight t
proof (induct ts arbitrary: t)
  case Nil thus case by simp
next
case (Cons u us) thus case by simp (metis le_trans)
qed

lemma sortedByWeight_insortTree:
[sortedByWeight ts; height t = 0; height ts = 0] ⟷
sortedByWeight (insortTree t ts)
by (induct ts rule: sortedByWeight.induct) auto

4.8 Pair of Minimal Symbols

The minima predicate expresses that two symbols \(a, b \in \text{alphabet} t\) have the lowest frequencies in the tree \(t\) and that \(\text{freq} t a \leq \text{freq} t b\). Minimal symbols need not be uniquely defined.

definition minima :: \(\alpha\) tree ⇒ \(\alpha\) ⇒ \(\alpha\) ⇒ bool where
minima t a b ≡
  a ∈ alphabet t ∧ b ∈ alphabet t ∧ a ≠ b ∧ freq t a ≤ freq t b
  ∧ (∀ c ∈ alphabet t. c ≠ a → c ≠ b → freq t c ≥ freq t a ∧ freq t c ≥ freq t b)

5 Formalization of the Textbook Proof

5.1 Four-Way Symbol Interchange Cost Lemma

If \(a\) and \(b\) are minima, and \(c\) and \(d\) are at the very bottom of the tree, then exchanging \(a\) and \(b\) with \(c\) and \(d\) doesn’t increase the cost. Graphically, we have

\[ \text{cost} \leq \text{cost} \]

This cost property is part of Knuth’s proof:
Let $V$ be an internal node of maximum distance from the root. If $w_1$ and $w_2$ are not the weights already attached to the children of $V$, we can interchange them with the values that are already there; such an interchange does not increase the weighted path length.

Lemma 16.2 in Cormen et al. [4, p. 389] expresses a similar property, which turns out to be a corollary of our cost property:

Let $C$ be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in the last bit.

**lemma** cost_swapFourSyms_le:
**assumes** consistent $t$ minima $t$ a b c $\in$ alphabet $t$ d $\in$ alphabet $t$
  depth $t$ c = height $t$ depth $t$ d = height $t$ c $\neq$ d
**shows** cost (swapFourSyms $t$ a b c d) $\leq$ cost $t$

**proof**

- **note** lens = swapFourSyms_def minima_def cost_swapSyms_le depth_le_height
- **show** thesis
  **proof** (cases $a \neq d \land b \neq c$)
  **case** True
  **show** thesis
  **proof** cases
  **assume** $a = c$
  **show** thesis
  **proof** cases
    **assume** $b = d$
    **thus** thesis using ($a = c$) True assms
    **by** (simp add: lens)
  next
    **assume** $b \neq d$
    **thus** thesis using ($a = c$) True assms
    **by** (simp add: lens)
  qed
  next
  **assume** $a \neq c$
  **show** thesis
  **proof** cases
  **assume** $b = d$
  **thus** thesis using ($a \neq c$) True assms
  **by** (simp add: lens)
  next
  **assume** $b \neq d$
  **have** cost (swapFourSyms $t$ a b c d) $\leq$ cost (swapSyms $t$ a c)
  **using** ($b \neq d$) ($a \neq c$) True assms **by** (clarsimp simp: lens)
  also have ... $\leq$ cost $t$ using ($b \neq d$) ($a \neq c$) True assms
  **by** (clarsimp simp: lens)
  **finally** **show** thesis.
  qed
5.2 Leaf Split Optimality Lemma

The tree \( \text{splitLeaf} \ t \ w_a \ a \ w_b \ b \) is optimum if \( t \) is optimum, under a few assumptions, notably that \( a \) and \( b \) are minima of the new tree and that \( \text{freq} \ t \ a = \text{w}a + \text{w}b \).

Graphically:

![Graphical representation of leaf split optimality](image)

This corresponds to the following fragment of Knuth’s proof:

Now it is easy to prove that the weighted path length of such a tree is minimized if and only if the tree with

\[
\begin{array}{c}
\text{w}_1 \\
\text{w}_2 \\
\end{array}
\]

replaced by

\[
\begin{array}{c}
\text{w}_1 + \text{w}_2 \\
\end{array}
\]

has minimum path length for the weights \( w_1, w_2, w_3, \ldots, w_m \).

We only need the “if” direction of Knuth’s equivalence. Lemma 16.3 in Cormen et al. [4, p. 391] expresses essentially the same property:

Let \( C \) be a given alphabet with frequency \( f[c] \) defined for each character \( c \in C \). Let \( x \) and \( y \) be two characters in \( C \) with minimum frequency. Let \( C' \) be the alphabet \( C \) with characters \( x, y \) removed and (new) character \( z \) added, so that \( C' = C - \{ x, y \} \cup \{ z \} \); define \( f \) for \( C' \) as for \( C \), except that \( f[z] = f[x] + f[y] \). Let \( T' \) be any tree representing an optimal prefix code for the alphabet \( C' \). Then the tree \( T \), obtained from \( T' \) by replacing the leaf node for \( z \) with an internal node having \( x \) and \( y \) as children, represents an optimal prefix code for the alphabet \( C \).
The proof is as follows: We assume that \( t \) has a cost less than or equal to that of any other comparable tree \( v \) and show that \( \text{splitLeaf} \ t \ w_a \ a \ w_b \ b \) has a cost less than or equal to that of any other comparable tree \( u \). By \text{exists_at_height} and \text{depth_height_imp_sibling_ne}, we know that some symbols \( c \) and \( d \) appear in sibling nodes at the very bottom of \( u \):

```
?  
\( c \) \( d \)
```

(The question mark is there to remind us that we know nothing specific about \( u \)’s structure.) From \( u \) we construct a new tree \( \text{swapFourSyms} \ u \ a \ b \ c \ d \) in which the minima \( a \) and \( b \) are siblings:

```
?  
\( a \) \( b \)
```

Merging \( a \) and \( b \) gives a tree comparable with \( t \), which we can use to instantiate \( v \) in the assumption:
With this instantiation, the proof is easy:

\[
\begin{align*}
\text{cost} \ (\text{splitLeaf} \ t \ a \ w \ b) &= \text{cost}\_\text{splitLeaf} \\
&= \text{cost} \ t + w_a + w_b \\
&\leq \underbrace{\text{cost} \ (\text{mergeSibling} \ (\text{swapFourSyms} \ u \ a \ b \ c \ d) \ a) + w_a + w_b \quad \text{(assumption)}} \\
&= \text{cost} \ (\text{swapFourSyms} \ u \ a \ b \ c \ d) \\
&\leq \text{cost} \ u.
\end{align*}
\]

In contrast, the proof in Cormen et al. is by contradiction: Essentially, they assume that there exists a tree \( u \) with a lower cost than \( \text{splitLeaf} \ t \ a \ w \ b \) and show that there exists a tree \( v \) with a lower cost than \( t \), contradicting the hypothesis that \( t \) is optimum. In place of \( \text{cost}\_\text{swapFourSyms}\_\text{le} \), they invoke their lemma 16.2, which is questionable since \( u \) is not necessarily optimum.\(^3\)

Our proof relies on the following lemma, which asserts that \( a \) and \( b \) are minima of \( u \).

**lemma** twice_freq_le_imp_minima:

\[
\forall c \in \text{alphabet} \ t. \ w_a \leq \text{freq} \ t c \land w_b \leq \text{freq} \ t c; \\
\text{alphabet} \ u = \text{alphabet} \ t \cup \{b\}; \ a \in \text{alphabet} \ u; \ a \neq b; \\
\text{freq} \ u = (\lambda c. \text{if } c = a \text{ then } w_a \text{ else if } c = b \text{ then } w_b \text{ else } \text{freq} \ t c); \\
w_a \leq w_b \implies \text{minima} \ u \ a \ b \\
\text{by (simp add: minima_def)}
\]

Now comes the key lemma.

**lemma** optimum_splitLeaf:

**assumes** consistent \( t \) optimum \( t \ a \in \text{alphabet} \ t \ b \not\in \text{alphabet} \ t \\
\text{freq} \ t a = w_a + w_b \ \forall c \in \text{alphabet} \ t. \ \text{freq} \ t c \geq w_a \land \text{freq} \ t c \geq w_b \\
w_a \leq w_b
\]

**shows** optimum \( (\text{splitLeaf} \ t \ w_a \ a \ w_b) \)

**proof** (unfold optimum_def, clarify)

\[
\begin{align*}
\text{fix} \ u \\
\text{let} \ t' &= \text{splitLeaf} \ t \ w_a \ a \ w_b \\
\text{assume} \ c_u: \text{consistent} \ u \\
\text{and} \ a_u: \text{alphabet} \ t' = \text{alphabet} \ u \\
\text{and} \ f_u: \text{freq} \ t' = \text{freq} \ u \\
\text{show} \ \text{cost} \ t' \leq \text{cost} \ u
\end{align*}
\]

**proof** (cases height \( t' = 0 \))

\(^3\)Thomas Cormen commented that this step will be clarified in the next edition of the book.
case True thus thesis by simp

next

case False

hence hu: height u > 0 using a u assms
  by (auto intro: height_gt_0_alphabet_eq_imp_height_gt_0)

have a b: a ∈ alphabet u using a u assms by fastforce

have a: a ≠ b using assms by blast

from exists_at_height [OF cu]

obtain c where ac: c ∈ alphabet u and dc: depth u c = height u ..

let d = sibling u c

have dc: d ≠ c using dc cu ac by (metis depth_height_imp_sibling_ne)

have ad: d ∈ alphabet u using dc
  by (rule sibling_ne_imp_sibling_in_alphabet)

have du: depth u d = height u using dc cu by simp

let u' = swapFourSyms u a b c d

have c u': consistent u' using cu by simp

have a: alphabet u' = alphabet u using a a b a c a d a u by simp

have f u': freq u' = freq u using a a b a c a d a f u by simp

have s a: sibling u' a = b using cu a a b a c a d
  by (rule sibling_swapFourSyms_when_4th_is_sibling)

let v = mergeSibling u'

have c v: consistent v using cu by simp

have v: alphabet v = alphabet t using s a cu a u, a a a u assms by auto

have f v: freq v = freq t
  using s a cu a u, f u f u [THEN sym] a u [THEN sym] assms
  by (simp add: freq_mergeSibling ext)

have cost t' = cost t + wa + wb using assms by simp

also have . . . ≤ cost v + wa + wb using c v a v f v optimum t
  by (simp add: optimum_def)

also have . . . = cost u'

  proof
    have v + freq u' a + freq u' (sibling u' a) = cost u'
      using cu s a assms by (subst cost_mergeSibling) auto
    moreover have wa = freq u' a wb = freq u' b
      using f u f u [THEN sym] assms by clarsimp
    ultimately show thesis using s a by simp
  qed

also have . . . ≤ cost u

  proof
    have minima u a b using a a f u assms
by (subst twice_freq_le_imp_minima) auto
with cu show thesis using a, a, d, d [THEN not_sym]
by (rule cost_swapFourSyms_le)
qed
finally show thesis .
qed
qed

5.3 Leaf Split Commutativity Lemma

A key property of Huffman’s algorithm is that once it has combined two lowest-
weight trees using uniteTrees, it doesn’t visit these trees ever again. This suggests
that splitting a leaf node before applying the algorithm should give the same
result as applying the algorithm first and splitting the leaf node afterward. The
diagram below illustrates the situation:

From the original forest (1), we can either run the algorithm (2a) and then split a
(3a) or split a (2b) and then run the algorithm (3b). Our goal is to show that trees
(3a) and (3b) are identical. Formally, we prove that

\[ \text{splitLeaf} (\text{huffman } ts) \ w _a \ a \ w _b \ b = \text{huffman} (\text{splitLeaf} _F \ ts \ w _a \ a \ w _b \ b) \]
when \( ts \) is consistent, \( a \in \text{alphabet}_F \ ts \), \( b \not\in \text{alphabet}_F \ ts \), and \( \text{freq}_F \ ts \ a = w_a + w_b \).

But before we can prove this commutativity lemma, we need to introduce a few simple lemmas.

**lemma** cachedWeight_splitLeaf [simp]:
\[
\text{cachedWeight} \ (\text{splitLeaf} \ t \ w_a \ w_b) = \text{cachedWeight} \ t
\]
by (case_tac \( t \)) simp

**lemma** splitLeaf_F_insortTree_when_in_alphabet_left [simp]:
\[
[a \in \text{alphabet} \ t; \ \text{consistent} \ t; \ a \not\in \text{alphabet}_F \ ts; \ \text{freq}_F \ ts \ a = w_a + w_b] \implies
\text{splitLeaf}_F \ (\text{insortTree} \ t \ ts) \ w_a \ w_b = \text{insortTree} \ (\text{splitLeaf} \ t \ w_a \ w_b) \ ts
\]
by (induct \( ts \)) simp

**lemma** splitLeaf_F_insortTree_when_in_alphabet_tail [simp]:
\[
[a \in \text{alphabet}_F \ ts; \ \text{consistent}_F \ ts; \ a \not\in \text{alphabet} \ t; \ \text{freq}_F \ ts \ a = w_a + w_b] \implies
\text{splitLeaf}_F \ (\text{insortTree} \ t \ ts) \ w_a \ w_b =
\text{insortTree} \ t \ (\text{splitLeaf}_F \ ts \ w_a \ w_b)
\]

**proof (induct \( ts \))**
  - case Nil thus case by simp
next
  - case (Cons \( u \) \( us \)) show case
    **proof (cases \( a \in \text{alphabet} \ u \))**
      - case True
        moreover hence \( a \not\in \text{alphabet}_F \ us \) using Cons by auto
        **ultimately show** thesis using Cons by auto
next
  - case False thus thesis using Cons by simp
qed

We are now ready to prove the commutativity lemma.

**lemma** splitLeaf_huffman_commute:
\[
[\text{consistent}_F \ ts; \ ts \neq []; \ a \in \text{alphabet}_F \ ts; \ \text{freq}_F \ ts \ a = w_a + w_b] \implies
\text{splitLeaf} \ (\text{huffman} \ ts) \ w_a \ w_b = \text{huffman} \ (\text{splitLeaf}_F \ ts \ w_a \ w_b)
\]

**proof (induct \( ts \) rule: huffman.induct)**
  - **BASE CASE 1: \( ts = [] \)**
    case 3 thus case by simp
next
  - **BASE CASE 2: \( ts = [t] \)**
    case (1 \( t \)) thus case by simp
next
  - **INDUCTION STEP: \( ts = t_1 \cdot t_2 \cdot ts \)**
    case (2 \( t_1 \) \( t_2 \) \( ts \))
    note hyps = 2
show case
proof (cases a ∈ alphabet t₁)
  case True
  moreover hence a ∉ alphabet t₂ a ∉ alphabet₆ ts using hyps by auto
  ultimately show thesis using hyps by (simp add: uniteTrees_def)
next
  case False
  note a₁ = False
  show thesis
  proof (cases a ∈ alphabet t₂)
    case True
    moreover hence a ∉ alphabet₆ ts using hyps by auto
    ultimately show thesis using a₁ hyps by (simp add: uniteTrees_def)
next
  case False
  thus thesis using a₁ hyps by simp
qed
qed

An important consequence of the commutativity lemma is that applying Huffman’s algorithm on a forest of the form

```
  c  w₁
      /   /
    a  b  d  ...
      w₃  w₄  w₅
```

gives the same result as applying the algorithm on the “flat” forest

```
  c  w₃
      /
    a  b  ...
      w₅
```

followed by splitting the leaf node a into two nodes a, b with frequencies w₃, w₅. The lemma effectively provides a way to flatten the forest at each step of the algorithm. Its invocation is implicit in the textbook proof.

5.4 Optimality Theorem

We are one lemma away from our main result.

**lemma** max_0_imp_0 [simp]:
(max x y = (0::nat)) = (x = 0 ∧ y = 0)
**by** auto
**Theorem** optimum_huffman:

\[
\text{consistent}_F\ ts; \ \text{height}_F\ ts = 0; \ \text{sortedByWeight}\ ts; \ ts \neq [] \implies \\
\text{optimum}\ (\text{huffman}\ ts)
\]

The input \(ts\) is assumed to be a nonempty consistent list of leaf nodes sorted by weight. The proof is by induction on the length of the forest \(ts\). Let \(ts\) be

\[
\begin{array}{cccccc}
a & b & c & d & \cdots & z \\
w_a & w_b & w_c & w_d & & w_z
\end{array}
\]

with \(w_a \leq w_b \leq w_c \leq w_d \leq \cdots \leq w_z\). If \(ts\) consists of a single leaf node, the node has cost 0 and is therefore optimum. If \(ts\) has length 2 or more, the first step of the algorithm leaves us with the term

\[
\begin{array}{cccccc}
\text{huffman} & c & a & d & \cdots & z \\
w_c & w_a + w_b & w_d & & w_z
\end{array}
\]

In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By \(\text{splitLeaf}_\text{huffman}\_\text{commute}\), the above tree equals

\[
\text{splitLeaf}\ \left(\text{huffman}\ \begin{array}{cccccc}
c & a & d & \cdots & z \\
w_c & w_a + w_b & w_d & & w_z
\end{array}\right) \ w_a \cdot w_b \cdot b.
\]

To prove that this tree is optimum, it suffices by \(\text{optimum}\_\text{splitLeaf}\) to show that

\[
\begin{array}{cccccc}
\text{huffman} & c & a & d & \cdots & z \\
w_c & w_a + w_b & w_d & w_z & &
\end{array}
\]

is optimum, which follows from the induction hypothesis.

**Proof** (induct ts rule: length_induct)

— Complete Induction Step

case (1 ts)

note hyps = 1

show case

proof (cases ts)

case Nil thus thesis using \(ts \neq []\) by fast

next

case (Cons \(t_d\ ts')\)

note \(ts = \text{Cons}\)

show thesis

proof (cases ts')
case Nil thus thesis using ts hyps by (simp add: optimum_def)

next
case (Cons t_a t_b)
note ts' = Cons
show thesis
proof (cases t_a)
case (Leaf w_a a)
note l_a = Leaf
show thesis
proof (cases t_b)
case (Leaf w_b b)
note l_b = Leaf
show thesis
proof
let us = insert (uniteTrees t_a t_b) ts''
let us' = insert (Leaf (w_a + w_b) a) ts''
let t_s = splitLeaf (huffman us') w_a w_b b
have e1: huffman ts = huffman us using ts' ts hyps by simp
have e2: huffman us = t_s using l_a l_b ts' ts hyps
  by (auto simp: splitLeaf_huffman_commute uniteTrees_def)

have optimum (huffman us') using l_a ts' ts hyps
  by (drule_tac x = us' in spec)
    (auto dest: sortedByWeight_Cons_imp_sortedByWeight
      simp: sortedByWeight_insert)

hence optimum t_s using l_a l_b ts' ts hyps
apply simp
apply (rule optimum_splitLeaf)
by (auto dest!: height_F_0_imp_Leaf_freqF_in_set
      sortedByWeight_Cons_imp_forall_weight_ge)

thus optimum (huffman ts) using e1 e2 by simp
qed
next
case InnerNode thus thesis using ts' ts hyps by simp
qed
next
case InnerNode thus thesis using ts' ts hyps by simp
qed
qed
qed

So what have we achieved? Assuming that our definitions really mean what
we intend them to mean, we established that our functional implementation of Huffman’s algorithm, when invoked properly, constructs a binary tree that represents an optimal prefix code for the specified alphabet and frequencies. Using Isabelle’s code generator [6], we can convert the Isabelle code into Standard ML, OCaml, or Haskell and use it in a real application.

As a side note, the optimum_huffman theorem assumes that the forest ts passed to huffman consists exclusively of leaf nodes. It is tempting to relax this restriction, by requiring instead that the forest ts has the lowest cost among forests of the same size. We would define optimality of a forest as follows:

\[
\text{optimum}_F \text{ts} \equiv (\forall \text{us. length ts} = \text{length us} \rightarrow \text{consistent}_F \text{us} \rightarrow \\
\text{alphabet}_F \text{ts} = \text{alphabet}_F \text{us} \rightarrow \text{freq}_F \text{ts} = \text{freq}_F \text{us} \rightarrow \\
\text{cost}_F \text{ts} \leq \text{cost}_F \text{us})
\]

with \(\text{cost}_F \left[\right] = 0\) and \(\text{cost}_F \left(t \cdot \text{ts}\right) = \text{cost} t + \text{cost}_F \text{ts}\). However, the modified proposition does not hold. A counterexample is the optimum forest

```
4
 / \ 5 5
 /   \
2 3 2 3
```

for which the algorithm constructs the tree

```
14
 / \
5 9
 /   \
2 2 4 4
     / \
    5 3
```

of greater cost than

```
14
 / \
6 8
 /   \
3 3 4 4
 /   \
2 2
```

6 Related Work

Laurent Théry’s Coq formalization of Huffman’s algorithm [14, 15] is an obvious yardstick for our work. It has a somewhat wider scope, proving among others the isomorphism between prefix codes and full binary trees. With 291 theorems, it is also much larger.

Théry identified the following difficulties in formalizing the textbook proof:

1. The leaf interchange process that brings the two minimal symbols together is tedious to formalize.

2. The sibling merging process requires introducing a new symbol for the merged node, which complicates the formalization.
3. The algorithm constructs the tree in a bottom-up fashion. While top-down procedures can usually be proved by structural induction, bottom-up procedures often require more sophisticated induction principles and larger invariants.

4. The informal proof relies on the notion of depth of a node. Defining this notion formally is problematic, because the depth can only be seen as a function if the tree is composed of distinct nodes.

To circumvent these difficulties, Théry introduced the ingenious concept of cover. A forest $ts$ is a cover of a tree $t$ if $t$ can be built from $ts$ by adding inner nodes on top of the trees in $ts$. The term “cover” is easier to understand if the binary trees are drawn with the root at the bottom of the page, like natural trees. Huffman’s algorithm is a refinement of the cover concept. The main proof consists in showing that the cost of $huffman ts$ is less than or equal to that of any other tree for which $ts$ is a cover. It relies on a few auxiliary definitions, notably an “ordered cover” concept that facilitates structural induction and a four-argument depth predicate (confusingly called $height$). Permutations also play a central role.

Incidentally, our experience suggests that the potential problems identified by Théry can be overcome more directly without too much work, leading to a simpler proof:

1. Formalizing the leaf interchange did not prove overly tedious. Among our 95 lemmas and theorems, 24 concern $swapLeaves$, $swapSyms$, and $swapFourSyms$.

2. The generation of a new symbol for the resulting node when merging two sibling nodes in $mergeSibling$ was trivially solved by reusing one of the two merged symbols.

3. The bottom-up nature of the tree construction process was addressed by using the length of the forest as the induction measure and by merging the two minimal symbols, as in Knuth’s proof.

4. By restricting our attention to consistent trees, we were able to define the $depth$ function simply and meaningfully.

7 Conclusion

The goal of most formal proofs is to increase our confidence in a result. In the case of Huffman’s algorithm, however, the chances that a bug would have gone unnoticed for the 56 years since its publication, under the scrutiny of leading computer scientists, seem extremely low; and the existence of a Coq proof should be sufficient to remove any remaining doubts.
The main contribution of this report has been to demonstrate that the textbook proof of Huffman’s algorithm can be elegantly formalized using a state-of-the-art theorem prover such as Isabelle/HOL. In the process, we uncovered a few minor snags in the proof given in Cormen et al. [4].

We also found that custom induction rules, in combination with suitable simplification rules, greatly help the automatic proof tactics, sometimes reducing 30-line proof scripts to one-liners. We successfully applied this approach for handling both the ubiquitous “datatype + wellformedness predicate” combination (a tree + consistent) and functions defined by sequential pattern matching (sibling and mergeSibling). Our experience suggests that such rules, which are uncommon in formalizations, are highly valuable and versatile. Moreover, Isabelle’s induction_schema and lexicographic_order tactics make these easy to prove.

Formalizing the proof of Huffman’s algorithm also led to a deeper understanding of this classic algorithm. Many of the lemmas, notably the leaf split commutativity lemma of Section 5.3, have not been found in the literature and express fundamental properties of the algorithm. Other discoveries didn’t find their way into the final proof. In particular, each step of the algorithm appears to preserve the invariant that the nodes in a forest are ordered by weight from left to right, bottom to top, as in the example below:

![Diagram of a tree](image)

It is not hard to prove formally that a tree exhibiting this property is optimum. On the other hand, proving that the algorithm preserves this invariant seems difficult—more difficult than formalizing the textbook proof—and remains a suggestion for future work.

A few other directions for future work suggest themselves. First, we could formalize some of our hypotheses, notably our restriction to full and consistent binary trees. The current formalization says nothing about the algorithm’s application for data compression, so the next step could be to extend the proof’s scope to cover encode/decode functions and show that full binary trees are isomorphic to prefix codes, as done in the Coq development. Independently, we could generalize the development to n-ary trees.
Acknowledgments

I am grateful to several people for their help in producing this report. Tobias Nipkow suggested that I cut my teeth on Huffman coding and discussed several (sometimes flawed) drafts of the proof. He also provided many insights into Isabelle, which led to considerable simplifications. Alexander Krauss answered all my Isabelle questions and helped me with the trickier proofs. Thomas Cormen and Donald Knuth were both gracious enough to discuss their proofs with me, and Donald Knuth also suggested a terminology change. Finally, Mark Summerfield and the anonymous reviewers of the corresponding journal paper proposed many textual improvements.

References


