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 14, 2024

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

*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 δ_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

$$\delta_c = \log_2 \tfrac{1}{p_c}, \qquad \text{where } p_c = \tfrac{w_c}{\sum_a w_a}.$$

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

$$p_{\mathrm{a}} = \tfrac{1}{2},\ p_{\mathrm{b}} = \tfrac{1}{4},\ p_{\mathrm{c}} = \tfrac{1}{8},\ p_{\mathrm{d}} = \tfrac{1}{8}.$$

The optimum lengths for the binary code words are all integers, namely

$$\delta_{\mathrm{a}} = 1,\ \delta_{\mathrm{b}} = 2,\ \delta_{\mathrm{c}} = 3,\ \delta_{\mathrm{d}} = 3,$$

and they are realized by the code

$$C_1 = \{\mathrm{a} \mapsto 0,\ \mathrm{b} \mapsto 10,\ \mathrm{c} \mapsto 110,\ \mathrm{d} \mapsto 111\}.$$

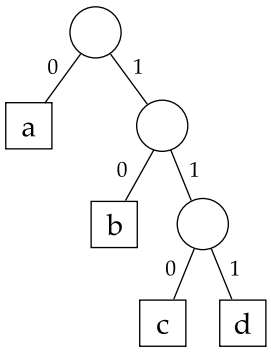
Encoding 'abacabad' produces the 14-bit code word 01001100100111. The code C_1 is optimum: No code that unambiguously encodes source symbols one at a time could do better than C_1 on the input 'abacabad'. In particular, with a fixed-length code such as

$$C_2 = \{\mathrm{a} \mapsto 00,\ \mathrm{b} \mapsto 01,\ \mathrm{c} \mapsto 10,\ \mathrm{d} \mapsto 11\}$$

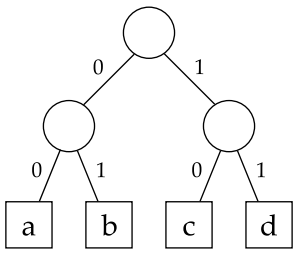
we need at least 16 bits to encode 'abacabad'.

1.2 Binary Trees

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



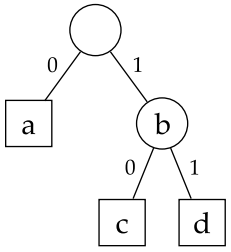
and



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

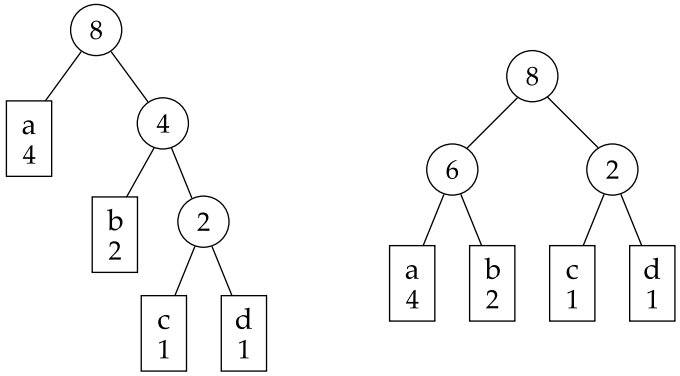
the symbol one node at a time; generate a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The generated sequence of 0s and 1s is the code word.

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.[1] Codes that have this property are called *prefix codes*. As an example of a code that does not have this property, consider the code associated with the tree



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

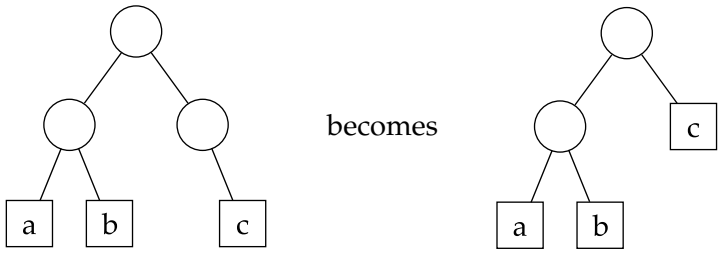
Each node in a code tree is assigned a *weight*. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. Code trees can be annotated with their weights:



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

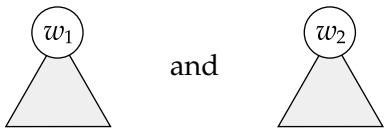
[1] Strictly speaking, there is another potential source of ambiguity. If the alphabet consists of a single symbol a, that symbol could be mapped to the empty code word, and then any string $aa \ldots a$ would map to the empty bit sequence, giving the decoder no way to recover the original string's length.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

child can advantageously be eliminated; for example,

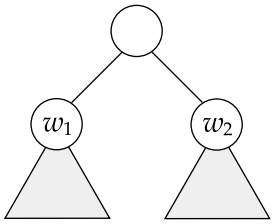


1.3 Huffman's Algorithm

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



with the lowest weights and replace them with the tree

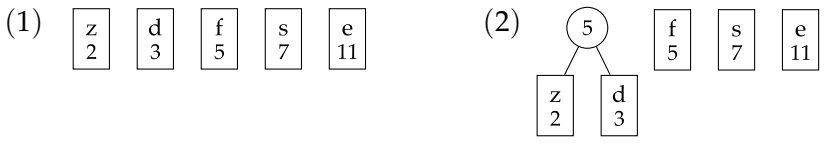


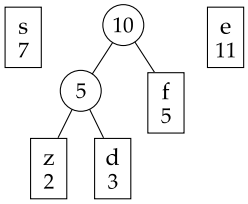
Repeat this process until only one tree is left.

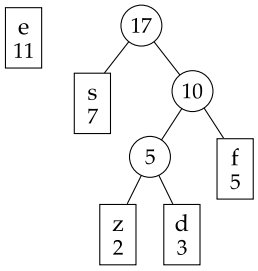
As an illustration, executing the algorithm for the frequencies

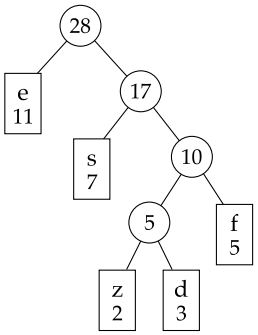
$$f_{\mathrm{d}} = 3,\ \ f_{\mathrm{e}} = 11,\ \ f_{\mathrm{f}} = 5,\ \ f_{\mathrm{s}} = 7,\ \ f_{\mathrm{z}} = 2$$

gives rise to the following sequence of states:



(3) 

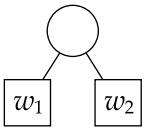
(4) 

(5) 

Tree (5) is an optimum tree for the given frequencies.

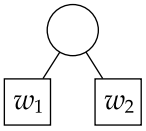
1.4 The Textbook Proof

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., $\sum_a w_a \delta_a$), 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
>
> 
>
>
> has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$.

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 document 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 a little under 100 lemmas and 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].

1.6 Head of the Theory File

The Isabelle theory starts in the standard way.

theory *Huffman*
imports *Main*
begin

We attach the *simp* attribute to some predefined lemmas to add them to the default set of simplification rules.

declare
Int_Un_distrib [*simp*]
*Int_Un_distrib*2 [*simp*]
*max.absorb*1 [*simp*]
*max.absorb*2 [*simp*]

2 Definition of Prefix Code Trees and Forests

2.1 Tree Type

A *prefix code tree* is a full binary tree in which leaf nodes are of the form *Leaf w a*, where *a* is a symbol and *w* is the frequency associated with *a*, and inner nodes are of the form *Node w* t_1 t_2, where t_1 and t_2 are the left and right subtrees and *w* caches the sum of the weights of t_1 and t_2. Prefix code trees are polymorphic on the symbol datatype *'a*.

datatype *'a tree* =
Leaf nat 'a
Node nat (*'a tree*) (*'a tree*)

2.2 Forest Type

The intermediate steps of Huffman's algorithm involve a list of prefix code trees, or *prefix code forest*.

type_synonym *'a forest* = *'a tree list*

2.3 Alphabet

The *alphabet* of a code tree is the set of symbols appearing in the tree's leaf nodes.

primrec *alphabet* :: *'a tree* ⇒ *'a set* **where**
alphabet (*Leaf w a*) = {*a*}
alphabet (*Node w* t_1 t_2) = *alphabet* t_1 ∪ *alphabet* t_2

For sets and predicates, Isabelle gives us the choice between inductive definitions (**inductive_set** and **inductive**) and recursive functions (**primrec**, **fun**, and **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.

primrec $alphabet_F :: 'a\ forest \Rightarrow 'a\ set$ **where**
$alphabet_F\ [] = \{\}$
$alphabet_F\ (t \cdot ts) = alphabet\ t \cup alphabet_F\ ts$

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

lemma *finite_alphabet*[*simp*]:
$finite\ (alphabet\ t)$
⟨*proof*⟩

lemma *exists_in_alphabet*:
$\exists a.\ a \in alphabet\ t$
⟨*proof*⟩

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 δ_a (the length of the *unique* code word for a) to be defined. Although this wellformedness property is not mentioned in algorithms textbooks [1, 4, 8], it is essential and appears as an assumption in many of our lemmas.

primrec $consistent :: 'a\ tree \Rightarrow bool$ **where**
$consistent\ (Leaf\ w\ a) = True$
$consistent\ (Node\ w\ t_1\ t_2) =$
$(alphabet\ t_1 \cap alphabet\ t_2 = \{\} \wedge consistent\ t_1 \wedge consistent\ t_2)$

primrec $consistent_F :: 'a\ forest \Rightarrow bool$ **where**
$consistent_F\ [] = True$
$consistent_F\ (t \cdot ts) =$
$(alphabet\ t \cap alphabet_F\ ts = \{\} \wedge consistent\ t \wedge consistent_F\ ts)$

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: $t = \mathit{Leaf}\ w\ b$.

INDUCTION STEP: $t = \mathit{Node}\ w\ t_1\ t_2$.

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*$_3$]:
$[\![\mathit{consistent}\ t;$
$\bigwedge w_b\ b\ a.\ P\ (\mathit{Leaf}\ w_b\ b)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$[\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};$
$a \in \mathit{alphabet}\ t_1;\ a \notin \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$P\ (\mathit{Node}\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$[\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};$
$a \notin \mathit{alphabet}\ t_1;\ a \in \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$P\ (\mathit{Node}\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$[\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};$
$a \notin \mathit{alphabet}\ t_1;\ a \notin \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$P\ (\mathit{Node}\ w\ t_1\ t_2)\ a]\!] \Longrightarrow$
$P\ t\ a$

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.

⟨*proof*⟩
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_induct_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 δ_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 do not 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.

primrec *depth* :: *'a tree* ⇒ *'a* ⇒ *nat* **where**
depth (*Leaf w b*) *a* = 0
depth (*Node w* t_1 t_2) *a* =
(*if a* ∈ *alphabet* t_1 *then depth* t_1 *a* + 1
else if a ∈ *alphabet* t_2 *then depth* t_2 *a* + 1
else 0)

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.

primrec *height* :: *'a tree* ⇒ *nat* **where**
height (*Leaf w a*) = 0
height (*Node w* t_1 t_2) = *max* (*height* t_1) (*height* t_2) + 1

primrec $height_F$ :: *'a forest* ⇒ *nat* **where**
$height_F$ [] = 0
$height_F$ (*t* · *ts*) = *max* (*height t*) ($height_F$ *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*:
depth t a ≤ *height t*
⟨*proof*⟩

lemma *exists_at_height*:
consistent t $\Longrightarrow \exists a \in$ *alphabet t*. *depth t a* = *height t*
⟨*proof*⟩

The following elimination rules help Isabelle's classical prover, notably the *auto* tactic. They are easy consequences of the inequation *depth t a* $\leq$ *height t*.

lemma *depth_max_heightE_left*[*elim*!]:
⟦*depth* t_1 *a* = *max* (*height* t_1) (*height* t_2);
⟦*depth* t_1 *a* = *height* t_1; *height* $t_1 \geq$ *height* t_2⟧ $\Longrightarrow$ *P*⟧ $\Longrightarrow$
P
⟨*proof*⟩

lemma *depth_max_heightE_right*[*elim*!]:
⟦*depth* t_2 *a* = *max* (*height* t_1) (*height* t_2);
⟦*depth* t_2 *a* = *height* t_2; *height* $t_2 \geq$ *height* t_1⟧ $\Longrightarrow$ *P*⟧ $\Longrightarrow$
P
⟨*proof*⟩

We also need the following lemma.

lemma *height_gt_0_alphabet_eq_imp_height_gt_0*:
assumes *height t* > 0 *consistent t alphabet t* = *alphabet u*
shows *height u* > 0
⟨*proof*⟩

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* :: *'a tree* ⇒ *'a* ⇒ *nat* **where**
freq (*Leaf w a*) *b* = (*if b* = *a then w else* 0)
freq (*Node w* t_1 t_2) *b* = *freq* t_1 *b* + *freq* t_2 *b*

primrec $freq_F$ :: *'a forest* ⇒ *'a* ⇒ *nat* **where**
$freq_F$ [] *b* = 0
$freq_F$ (*t* · *ts*) *b* = *freq t b* + $freq_F$ *ts b*

Alphabet and symbol frequencies are intimately related. Simplification rules ensure that sums of the form *freq* t_1 *a* + *freq* t_2 *a* collapse to a single term when we know which tree *a* belongs to.

lemma *notin_alphabet_imp_freq_0*[*simp*]:

$a \notin \mathit{alphabet}\ t \Longrightarrow \mathit{freq}\ t\ a = 0$
⟨*proof*⟩

lemma *notin_alphabet*$_F$*_imp_freq*$_F$*_0*[*simp*]:
$a \notin \mathit{alphabet}_F\ ts \Longrightarrow \mathit{freq}_F\ ts\ a = 0$
⟨*proof*⟩

lemma *freq_0_right*[*simp*]:
$[\![\mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};\ a \in \mathit{alphabet}\ t_1]\!] \Longrightarrow \mathit{freq}\ t_2\ a = 0$
⟨*proof*⟩

lemma *freq_0_left*[*simp*]:
$[\![\mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};\ a \in \mathit{alphabet}\ t_2]\!] \Longrightarrow \mathit{freq}\ t_1\ a = 0$
⟨*proof*⟩

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 *height*$_F$*_0_imp_Leaf_freq*$_F$*_in_set*:
$[\![\mathit{consistent}_F\ ts;\ \mathit{height}_F\ ts = 0;\ a \in \mathit{alphabet}_F\ ts]\!] \Longrightarrow$
$\mathit{Leaf}\ (\mathit{freq}_F\ ts\ a)\ a \in \mathit{set}\ ts$
⟨*proof*⟩

2.8 Weight

The *weight* function returns the weight of a tree. In the *Node* 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* :: *'a tree* ⇒ *nat* **where**
$\mathit{weight}\ (\mathit{Leaf}\ w\ a) = w$
$\mathit{weight}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2$

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

lemma *weight_eq_Sum_freq*:
$\mathit{consistent}\ t \Longrightarrow \mathit{weight}\ t = \sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a$
⟨*proof*⟩
The assumption *consistent t* is not necessary, but it simplifies the proof by letting

us invoke the lemma *sum.union_disjoint*:

$$[\![\mathit{finite}\ A;\ \mathit{finite}\ B;\ A \cap B = \{\}]\!] \Longrightarrow \sum_{x \in A} g\ x + \sum_{x \in B} g\ x = \sum_{x \in A \cup B} g\ x.$$

2.9 Cost

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

primrec *cost* :: *'a tree* ⇒ *nat* **where**
cost (*Leaf w a*) = 0
cost (*Node w* t_1 t_2) = *weight* t_1 + *cost* t_1 + *weight* t_2 + *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 *weight* (*Node w* t_1 t_2) = *weight* t_1 + *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 $\Longrightarrow$ *cost t* = $\sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a$

The proof is by structural induction on t. If t = *Leaf w b*, both sides of the equation simplify to 0. This leaves the case t = *Node w* t_1 t_2. Let A, A_1, and A_2 stand for

alphabet t, *alphabet* t_1, and *alphabet* t_2, respectively. We have

$$
\begin{aligned}
& \mathit{cost}\ t \\
= \quad & \qquad \text{(definition of } \mathit{cost}\text{)} \\
& \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2 \\
= \quad & \qquad \text{(induction hypothesis)} \\
& \mathit{weight}\ t_1 + \textstyle\sum_{a \in A_1} \mathit{freq}\ t_1\ a \times \mathit{depth}\ t_1\ a + \\
& \mathit{weight}\ t_2 + \textstyle\sum_{a \in A_2} \mathit{freq}\ t_2\ a \times \mathit{depth}\ t_2\ a \\
= \quad & \qquad \text{(definition of } \mathit{depth}\text{, consistency)} \\
& \mathit{weight}\ t_1 + \textstyle\sum_{a \in A_1} \mathit{freq}\ t_1\ a \times (\mathit{depth}\ t\ a - 1) + \\
& \mathit{weight}\ t_2 + \textstyle\sum_{a \in A_2} \mathit{freq}\ t_2\ a \times (\mathit{depth}\ t\ a - 1) \\
= \quad & \qquad \text{(distributivity of } \times \text{ and } \textstyle\sum \text{ over } -\text{)} \\
& \mathit{weight}\ t_1 + \textstyle\sum_{a \in A_1} \mathit{freq}\ t_1\ a \times \mathit{depth}\ t\ a - \sum_{a \in A_1} \mathit{freq}\ t_1\ a + \\
& \mathit{weight}\ t_2 + \textstyle\sum_{a \in A_2} \mathit{freq}\ t_2\ a \times \mathit{depth}\ t\ a - \sum_{a \in A_2} \mathit{freq}\ t_2\ a \\
= \quad & \qquad (\mathit{weight\_eq\_Sum\_freq}) \\
& \textstyle\sum_{a \in A_1} \mathit{freq}\ t_1\ a \times \mathit{depth}\ t\ a + \sum_{a \in A_2} \mathit{freq}\ t_2\ a \times \mathit{depth}\ t\ a \\
= \quad & \qquad \text{(definition of } \mathit{freq}\text{, consistency)} \\
& \textstyle\sum_{a \in A_1} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a + \sum_{a \in A_2} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a \\
= \quad & \qquad (\mathit{sum.union\_disjoint}\text{, consistency}) \\
& \textstyle\sum_{a \in A_1 \cup A_2} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a \\
= \quad & \qquad \text{(definition of } \mathit{alphabet}\text{)} \\
& \textstyle\sum_{a \in A} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a.
\end{aligned}
$$

The structured proof closely follows this argument.

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

lemma *height_0_imp_cost_0*[*simp*]:
$\mathit{height}\ t = 0 \Longrightarrow \mathit{cost}\ t = 0$
⟨*proof*⟩

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* :: *'a tree* ⇒ *bool* **where**
optimum t =
$(\forall u.\ \mathit{consistent}\ u \longrightarrow \mathit{alphabet}\ t = \mathit{alphabet}\ u \longrightarrow \mathit{freq}\ t = \mathit{freq}\ u \longrightarrow$
$\mathit{cost}\ t \leq \mathit{cost}\ u)$

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.

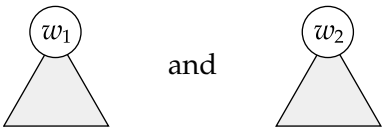
primrec *cachedWeight* :: *'a tree* ⇒ *nat* **where**
cachedWeight (*Leaf w a*) = *w*
cachedWeight (*Node w t*$_1$ *t*$_2$) = *w*

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

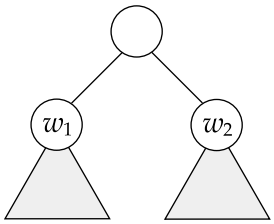
lemma *height_0_imp_cachedWeight_eq_weight*[*simp*]:
height t = 0 ⟹ *cachedWeight t* = *weight t*
⟨*proof*⟩

3.2 Tree Union

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



and returns the tree



definition *uniteTrees* :: *'a tree* ⇒ *'a tree* ⇒ *'a tree* **where**
uniteTrees t$_1$ *t*$_2$ = *Node* (*cachedWeight t*$_1$ + *cachedWeight t*$_2$) *t*$_1$ *t*$_2$

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*]:
alphabet (*uniteTrees t*$_1$ *t*$_2$) = *alphabet t*$_1$ ∪ *alphabet t*$_2$
⟨*proof*⟩

lemma *consistent_uniteTrees*[*simp*]:

$[\![consistent\ t_1;\ consistent\ t_2;\ alphabet\ t_1 \cap alphabet\ t_2 = \{\}]\!] \Longrightarrow$
$consistent\ (uniteTrees\ t_1\ t_2)$
$\langle proof \rangle$

lemma *freq_uniteTrees*[*simp*]:
$freq\ (uniteTrees\ t_1\ t_2)\ a = freq\ t_1\ a + freq\ t_2\ a$
$\langle proof \rangle$

3.3 Ordered Tree Insertion

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

primrec *insortTree* :: *'a tree* ⇒ *'a forest* ⇒ *'a forest* **where**
$insortTree\ u\ [] = [u]$
$insortTree\ u\ (t \cdot ts) =$
$(if\ cachedWeight\ u \leq cachedWeight\ t\ then\ u \cdot t \cdot ts\ else\ t \cdot insortTree\ u\ ts)$

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

lemma *length_insortTree*[*simp*]:
$length\ (insortTree\ t\ ts) = length\ ts + 1$
$\langle proof \rangle$

lemma *insortTree_ne_Nil*[*simp*]:
$insortTree\ t\ ts \neq []$
$\langle proof \rangle$

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*]:
$alphabet_F\ (insortTree\ t\ ts) = alphabet\ t \cup alphabet_F\ ts$
$\langle proof \rangle$

lemma $consistent_F$*_insortTree*[*simp*]:
$consistent_F\ (insortTree\ t\ ts) = consistent_F\ (t \cdot ts)$
$\langle proof \rangle$

lemma $freq_F$*_insortTree*[*simp*]:
$freq_F\ (insortTree\ t\ ts) = (\lambda a.\ freq\ t\ a + freq_F\ ts\ a)$
$\langle proof \rangle$

lemma $height_F$*_insortTree*[*simp*]:
$height_F\ (insortTree\ t\ ts) = max\ (height\ t)\ (height_F\ ts)$

⟨*proof*⟩

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* :: *'a forest* ⇒ *'a tree* **where**
huffman [*t*] = *t*
huffman (*t*$_1$ · *t*$_2$ · *ts*) = *huffman* (*insortTree* (*uniteTrees* *t*$_1$ *t*$_2$) *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 recomputed the weight each time it is needed, the complexity would remain quadratic, but with a larger constant. Using a binary search in *insortTree*, the corresponding 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*
⟨*proof*⟩

theorem *consistent_huffman*[*simp*]:
⟦*consistent*$_F$ *ts*; *ts* ≠ []⟧ ⟹ *consistent* (*huffman ts*)
⟨*proof*⟩

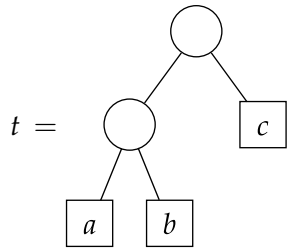
theorem *freq_huffman*[*simp*]:
ts ≠ [] ⟹ *freq* (*huffman ts*) *a* = *freq*$_F$ *ts a*
⟨*proof*⟩

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 *sibling t a* ≠ *a* if and only if *a* has a sibling. As an illustration, we have *sibling t a* = *b*, *sibling t b* = *a*, and

sibling t c = *c* for the tree



fun *sibling* :: *'a tree* ⇒ *'a* ⇒ *'a* **where**
sibling (*Leaf* w_b *b*) *a* = *a*
sibling (*Node w* (*Leaf* w_b *b*) (*Leaf* w_c *c*)) *a* =
(*if a* = *b then c else if a* = *c then b else a*)
sibling (*Node w* t_1 t_2) *a* =
(*if a* ∈ *alphabet* t_1 *then sibling* t_1 *a*
else if a ∈ *alphabet* t_2 *then sibling* t_2 *a*
else a)

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

lemma *notin_alphabet_imp_sibling_id*[*simp*]:
a ∉ *alphabet t* ⟹ *sibling t a* = *a*
⟨*proof*⟩

lemma *height_0_imp_sibling_id*[*simp*]:
height t = 0 ⟹ *sibling t a* = *a*
⟨*proof*⟩

lemma *height_gt_0_in_alphabet_imp_sibling_left*[*simp*]:
⟦*height* t_1 > 0; *a* ∈ *alphabet* t_1⟧ ⟹
sibling (*Node w* t_1 t_2) *a* = *sibling* t_1 *a*
⟨*proof*⟩

lemma *height_gt_0_in_alphabet_imp_sibling_right*[*simp*]:
⟦*height* t_2 > 0; *a* ∈ *alphabet* t_1⟧ ⟹
sibling (*Node w* t_1 t_2) *a* = *sibling* t_1 *a*
⟨*proof*⟩

lemma *height_gt_0_notin_alphabet_imp_sibling_left*[*simp*]:
⟦*height* t_1 > 0; *a* ∉ *alphabet* t_1⟧ ⟹
sibling (*Node w* t_1 t_2) *a* = *sibling* t_2 *a*
⟨*proof*⟩

lemma *height_gt_0_notin_alphabet_imp_sibling_right*[*simp*]:

$[\![height\ t_2 > 0;\ a \notin alphabet\ t_1]\!] \Longrightarrow$
$sibling\ (Node\ w\ t_1\ t_2)\ a = sibling\ t_2\ a$
⟨*proof*⟩

lemma *either_height_gt_0_imp_sibling*[*simp*]:
$height\ t_1 > 0 \vee height\ t_2 > 0 \Longrightarrow$
$sibling\ (Node\ w\ t_1\ t_2)\ a =$
$(if\ a \in alphabet\ t_1\ then\ sibling\ t_1\ a\ else\ sibling\ t_2\ a)$
⟨*proof*⟩

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

lemma *in_alphabet_imp_sibling_in_alphabet*:
$a \in alphabet\ t \Longrightarrow sibling\ t\ a \in alphabet\ t$
⟨*proof*⟩

lemma *sibling_ne_imp_sibling_in_alphabet*:
$sibling\ t\ a \neq a \Longrightarrow sibling\ t\ a \in alphabet\ t$
⟨*proof*⟩

The default induction rule for *sibling* distinguishes four cases.

BASE CASE: $t = Leaf\ w\ b$.

INDUCTION STEP 1: $t = Node\ w\ (Leaf\ w_b\ b)\ (Leaf\ w_c\ c)$.

INDUCTION STEP 2: $t = Node\ w\ (Node\ w_1\ t_{11}\ t_{12})\ t_2$.

INDUCTION STEP 3: $t = Node\ w\ t_1\ (Node\ w_2\ t_{21}\ t_{22})$.

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 *Node* constructors in the last two cases reduce readability. Third, under 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: $t = Node\ w\ (Leaf\ w_b\ b)\ (Leaf\ w_c\ c)$ with $b \neq c$.

INDUCTION STEP 2: $t = Node\ w\ t_1\ t_2$ 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 step$_1$ *step*$_{21}$ *step*$_{22}$ *step*$_{23}$]:
⟦*consistent t*;
⋀*w b a*. *P* (*Leaf w b*) *a*;
⋀*w w*$_b$ *b w*$_c$ *c a*. *b* ≠ *c* ⟹ *P* (*Node w* (*Leaf w*$_b$ *b*) (*Leaf w*$_c$ *c*)) *a*;
⋀*w t*$_1$ *t*$_2$ *a*.
⟦*consistent t*$_1$; *consistent t*$_2$; *alphabet t*$_1$ ∩ *alphabet t*$_2$ = {};
height t$_1$ > 0 ∨ *height t*$_2$ > 0; *a* ∈ *alphabet t*$_1$;
sibling t$_1$ *a* ∈ *alphabet t*$_1$; *a* ∉ *alphabet t*$_2$;
sibling t$_1$ *a* ∉ *alphabet t*$_2$; *P t*$_1$ *a*⟧ ⟹
P (*Node w t*$_1$ *t*$_2$) *a*;
⋀*w t*$_1$ *t*$_2$ *a*.
⟦*consistent t*$_1$; *consistent t*$_2$; *alphabet t*$_1$ ∩ *alphabet t*$_2$ = {};
height t$_1$ > 0 ∨ *height t*$_2$ > 0; *a* ∉ *alphabet t*$_1$;
sibling t$_2$ *a* ∉ *alphabet t*$_1$; *a* ∈ *alphabet t*$_2$;
sibling t$_2$ *a* ∈ *alphabet t*$_2$; *P t*$_2$ *a*⟧ ⟹
P (*Node w t*$_1$ *t*$_2$) *a*;
⋀*w t*$_1$ *t*$_2$ *a*.
⟦*consistent t*$_1$; *consistent t*$_2$; *alphabet t*$_1$ ∩ *alphabet t*$_2$ = {};
height t$_1$ > 0 ∨ *height t*$_2$ > 0; *a* ∉ *alphabet t*$_1$; *a* ∉ *alphabet t*$_2$⟧ ⟹
P (*Node w t*$_1$ *t*$_2$) *a*⟧ ⟹
P t a
⟨*proof*⟩

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

lemma *sibling_sibling_id*[*simp*]:
consistent t ⟹ *sibling t* (*sibling t a*) = *a*
⟨*proof*⟩

lemma *sibling_reciprocal*:
⟦*consistent t*; *sibling t a* = *b*⟧ ⟹ *sibling t b* = *a*
⟨*proof*⟩

lemma *depth_height_imp_sibling_ne*:
⟦*consistent t*; *depth t a* = *height t*; *height t* > 0; *a* ∈ *alphabet t*⟧ ⟹
sibling t a ≠ *a*
⟨*proof*⟩

lemma *depth_sibling*[*simp*]:
consistent t ⟹ *depth t* (*sibling t a*) = *depth t a*

⟨*proof*⟩

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 does not invoke *swapLeaves*.

primrec *swapLeaves* :: *'a tree* ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a* ⇒ *'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 (*Node w* t_1 t_2) w_a *a* w_b *b* =
  Node w (*swapLeaves* t_1 w_a *a* w_b *b*) (*swapLeaves* t_2 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 ∉ *alphabet t* ⟹ *swapLeaves t w a w' a* = *t*
⟨*proof*⟩

lemma *swapLeaves_id*[*simp*]:
consistent t ⟹ *swapLeaves t* (*freq t a*) *a* (*freq t a*) *a* = *t*
⟨*proof*⟩

The alphabet, consistency, symbol depths, height, and symbol frequencies of the tree *swapLeaves t* w_a *a* w_b *b* can be related to the homologous properties of t.

lemma *alphabet_swapLeaves*:
alphabet (*swapLeaves t* w_a *a* w_b *b*) =
  (*if a* ∈ *alphabet t then*
    if b ∈ *alphabet t then alphabet t else* (*alphabet t* − {*a*}) ∪ {*b*}
   else
    if b ∈ *alphabet t then* (*alphabet t* − {*b*}) ∪ {*a*} *else alphabet t*)
⟨*proof*⟩

lemma *consistent_swapLeaves*[*simp*]:
consistent t ⟹ *consistent* (*swapLeaves t* w_a *a* w_b *b*)
⟨*proof*⟩

lemma *depth_swapLeaves_neither*[*simp*]:
⟦*consistent t*; *c* ≠ *a*; *c* ≠ *b*⟧ ⟹ *depth* (*swapLeaves t* w_a *a* w_b *b*) *c* = *depth t c*

⟨*proof*⟩

lemma *height_swapLeaves*[*simp*]:
height (*swapLeaves t* w_a *a* w_b *b*) = *height t*
⟨*proof*⟩

lemma *freq_swapLeaves*[*simp*]:
⟦*consistent t*; *a* ≠ *b*⟧ ⟹
freq (*swapLeaves t* w_a *a* w_b *b*) =
  (λ*c*. *if c* = *a then if b* ∈ *alphabet t then* w_a *else* 0
    else if c = *b then if a* ∈ *alphabet t then* w_b *else* 0
    else freq t c)
⟨*proof*⟩

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

$$weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ a = weight\ t + w_b$$

rather than the more conventional

$$weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) = weight\ t + w_b - freq\ t\ a.$$

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 *freq t a* ≤ *weight t* (an easy consequence of *weight_eq_Sum_freq*), and then apply the *arith* tactic, but it is much simpler to use the first equation and stay with *simp* and *auto*. Another option would be to use integers instead of natural numbers.

lemma *weight_swapLeaves*:
⟦*consistent t*; *a* ≠ *b*⟧ ⟹
if a ∈ *alphabet t then*
 if b ∈ *alphabet t then*
  weight (*swapLeaves t* w_a *a* w_b *b*) + *freq t a* + *freq t b* =
    weight t + w_a + w_b
 else
  weight (*swapLeaves t* w_a *a* w_b *b*) + *freq t a* = *weight t* + w_b
else
 if b ∈ *alphabet t then*
  weight (*swapLeaves t* w_a *a* w_b *b*) + *freq t b* = *weight t* + w_a
 else
  weight (*swapLeaves t* w_a *a* w_b *b*) = *weight t*
⟨*proof*⟩

lemma *cost_swapLeaves*:
⟦*consistent t*; *a* ≠ *b*⟧ ⟹

if $a \in$ *alphabet t then*
 if $b \in$ *alphabet t then*
  cost (*swapLeaves t* w_a *a* w_b *b*) + *freq t a* × *depth t a*
  + *freq t b* × *depth t b* =
    cost t + w_a × *depth t b* + w_b × *depth t a*
 else
  cost (*swapLeaves t* w_a *a* w_b *b*) + *freq t a* × *depth t a* =
    cost t + w_b × *depth t a*
else
 if $b \in$ *alphabet t then*
  cost (*swapLeaves t* w_a *a* w_b *b*) + *freq t b* × *depth t b* =
    cost t + w_a × *depth t b*
 else
  cost (*swapLeaves t* w_a *a* w_b *b*) = *cost t*
⟨*proof*⟩

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*]:
⟦*consistent t*; *sibling t b* ≠ *b*; *a* ≠ *b*⟧ ⟹
sibling (*swapLeaves t* w_a *a* w_s (*sibling t b*)) *a* = *b*
⟨*proof*⟩

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* :: *'a tree* ⇒ *'a* ⇒ *'a* ⇒ *'a 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*
⟨*proof*⟩

lemma *alphabet_swapSyms*[*simp*]:
⟦*a* ∈ *alphabet t*; *b* ∈ *alphabet t*⟧ ⟹ *alphabet* (*swapSyms t a b*) = *alphabet t*
⟨*proof*⟩

lemma *consistent_swapSyms*[*simp*]:
consistent t ⟹ *consistent* (*swapSyms t a b*)
⟨*proof*⟩

lemma *depth_swapSyms_neither*[*simp*]:
⟦*consistent t*; *c* ≠ *a*; *c* ≠ *b*⟧ ⟹
depth (*swapSyms t a b*) *c* = *depth t c*
⟨*proof*⟩

lemma *freq_swapSyms*[*simp*]:
⟦*consistent t*; *a* ∈ *alphabet t*; *b* ∈ *alphabet t*⟧ ⟹
freq (*swapSyms t a b*) = *freq t*
⟨*proof*⟩

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*⟩

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 *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*⟩

As stated earlier, "If Astrid exchanges her house with Bernard's neighbor, Bernard becomes Astrid's new neighbor."

lemma *sibling_swapSyms_sibling*[*simp*]:
⟦*consistent t*; *sibling t b* ≠ *b*; *a* ≠ *b*⟧ ⟹
sibling (*swapSyms t a* (*sibling t b*)) *a* = *b*
⟨*proof*⟩

"If Astrid exchanges her house with Bernard, Astrid becomes Bernard's old neighbor's new neighbor."

lemma *sibling_swapSyms_other_sibling*[*simp*]:
⟦*consistent t*; *sibling t b* ≠ *a*; *sibling t b* ≠ *b*; *a* ≠ *b*⟧ ⟹
sibling (*swapSyms t a b*) (*sibling t b*) = *a*
⟨*proof*⟩

4.4 Four-Way Symbol Interchange

The *swapSyms* function exchanges two symbols *a* and *b*. We use it to define the four-way symbol interchange function *swapFourSyms*, which takes four symbols *a*, *b*, *c*, *d* with *a* ≠ *b* and *c* ≠ *d*, and exchanges them so that *a* and *b* occupy

c and d's positions.

A naive definition of this function would be

$$swapFourSyms\ t\ a\ b\ c\ d = swapSyms\ (swapSyms\ t\ a\ c)\ b\ d.$$

This definition fails in the face of aliasing: If $a = d$, but $b \neq c$, then *swapFourSyms a b c d* would leave a in b's position.[2]

definition *swapFourSyms* :: *'a tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* ⇒ *'a* ⇒ *'a tree* **where**
swapFourSyms t a b c d =
(*if a* = *d then swapSyms t b c*
else if b = *c then swapSyms t a d*
else swapSyms (*swapSyms t a c*) *b d*)

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

lemma *alphabet_swapFourSyms*[*simp*]:
⟦*a* ∈ *alphabet t*; *b* ∈ *alphabet t*; *c* ∈ *alphabet t*; *d* ∈ *alphabet t*⟧ ⟹
alphabet (*swapFourSyms t a b c d*) = *alphabet t*
⟨*proof*⟩

lemma *consistent_swapFourSyms*[*simp*]:
consistent t ⟹ *consistent* (*swapFourSyms t a b c d*)
⟨*proof*⟩

lemma *freq_swapFourSyms*[*simp*]:
⟦*consistent t*; *a* ∈ *alphabet t*; *b* ∈ *alphabet t*; *c* ∈ *alphabet t*;
d ∈ *alphabet t*⟧ ⟹
freq (*swapFourSyms t a b c d*) = *freq t*
⟨*proof*⟩

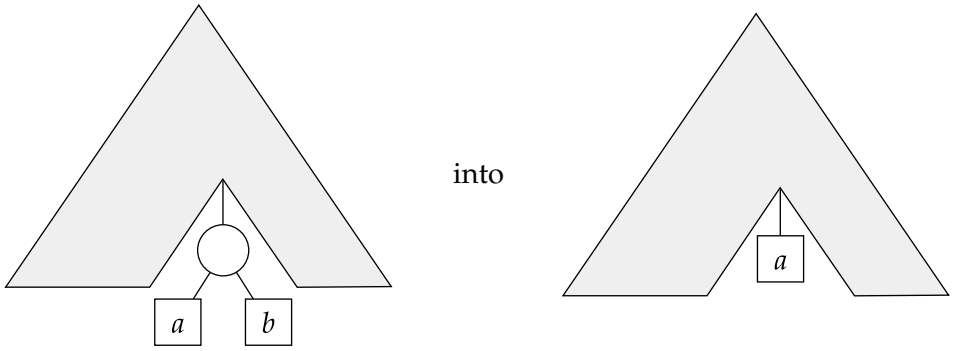
"If Astrid and Bernard exchange their houses with Carmen and her neighbor, Astrid and Bernard will now be neighbors."

lemma *sibling_swapFourSyms_when_4th_is_sibling*:
assumes *consistent t a* ∈ *alphabet t b* ∈ *alphabet t c* ∈ *alphabet t*
a ≠ *b sibling t c* ≠ *c*
shows *sibling* (*swapFourSyms t a b c* (*sibling t c*)) *a* = *b*
⟨*proof*⟩

[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.

4.5 Sibling Merge

Given a symbol a, the *mergeSibling* function transforms the tree



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.

fun *mergeSibling* :: *'a tree* ⇒ *'a* ⇒ *'a tree* **where**
mergeSibling (*Leaf* w_b b) a = *Leaf* w_b b
mergeSibling (*Node* w (*Leaf* w_b b) (*Leaf* w_c c)) a =
  (*if* $a = b \vee a = c$ *then Leaf* ($w_b + w_c$) a
  else Node w (*Leaf* w_b b) (*Leaf* w_c c))
mergeSibling (*Node* w t_1 t_2) a =
  Node w (*mergeSibling* t_1 a) (*mergeSibling* t_2 a)

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

lemmas *mergeSibling_induct_consistent* = *sibling_induct_consistent*

The properties of *mergeSibling* echo those of *sibling*. Like with *sibling*, simplification rules are crucial.

lemma *notin_alphabet_imp_mergeSibling_id*[*simp*]:
$a \notin$ *alphabet* $t \Longrightarrow$ *mergeSibling* t $a = t$
⟨*proof*⟩

lemma *height_gt_0_imp_mergeSibling_left*[*simp*]:
height $t_1 > 0 \Longrightarrow$
mergeSibling (*Node* w t_1 t_2) a =
  Node w (*mergeSibling* t_1 a) (*mergeSibling* t_2 a)
⟨*proof*⟩

lemma *height_gt_0_imp_mergeSibling_right*[*simp*]:
height $t_2 > 0 \Longrightarrow$
mergeSibling (*Node* w t_1 t_2) a =

Node w (*mergeSibling* t_1 *a*) (*mergeSibling* t_2 *a*)
⟨*proof*⟩

lemma *either_height_gt_0_imp_mergeSibling*[*simp*]:
height $t_1 > 0 \vee$ *height* $t_2 > 0 \Longrightarrow$
mergeSibling (*Node w* t_1 t_2) *a* =
Node w (*mergeSibling* t_1 *a*) (*mergeSibling* t_2 *a*)
⟨*proof*⟩

lemma *alphabet_mergeSibling*[*simp*]:
⟦*consistent t*; *a* ∈ *alphabet t*⟧ ⟹
alphabet (*mergeSibling t a*) = (*alphabet t* − {*sibling t a*}) ∪ {*a*}
⟨*proof*⟩

lemma *consistent_mergeSibling*[*simp*]:
consistent t ⟹ *consistent* (*mergeSibling t a*)
⟨*proof*⟩

lemma *freq_mergeSibling*:
⟦*consistent t*; *a* ∈ *alphabet t*; *sibling t a* ≠ *a*⟧ ⟹
freq (*mergeSibling t a*) =
(λ*c*. *if c* = *a then freq t a* + *freq t* (*sibling t a*)
else if c = *sibling t a then* 0
else freq t c)
⟨*proof*⟩

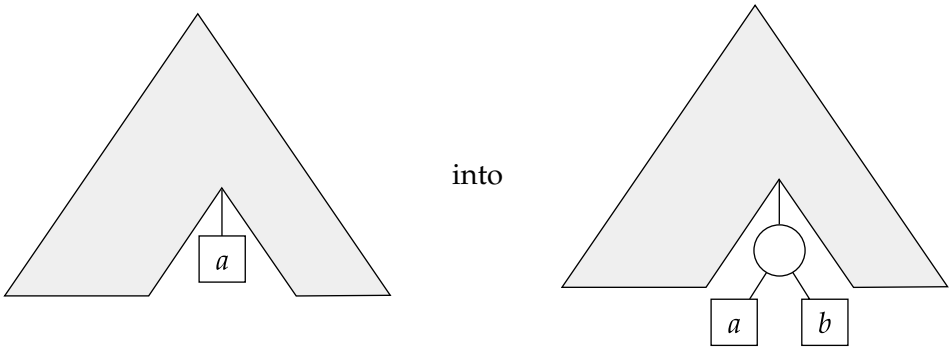
lemma *weight_mergeSibling*[*simp*]:
weight (*mergeSibling t a*) = *weight t*
⟨*proof*⟩

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*:
⟦*consistent t*; *sibling t a* ≠ *a*⟧ ⟹
cost (*mergeSibling t a*) + *freq t a* + *freq t* (*sibling t a*) = *cost t*
⟨*proof*⟩

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



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 $freq\ t\ a = w_a + w_b$.

primrec *splitLeaf* :: *'a tree* ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a* ⇒ *'a tree* **where**
splitLeaf (*Leaf* w_c c) w_a a w_b b =
(*if* $c = a$ *then Node* w_c (*Leaf* w_a a) (*Leaf* w_b b) *else Leaf* w_c c)
splitLeaf (*Node* w t_1 t_2) w_a a w_b b =
Node w (*splitLeaf* t_1 w_a a w_b b) (*splitLeaf* t_2 w_a a w_b b)

primrec $splitLeaf_F$ :: *'a forest* ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a* ⇒ *'a forest* **where**
$splitLeaf_F$ [] w_a a w_b b = []
$splitLeaf_F$ (t · ts) w_a a w_b b =
  splitLeaf t w_a a w_b b · $splitLeaf_F$ 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*]:
$a \notin alphabet\ t \Longrightarrow splitLeaf\ t\ w_a\ a\ w_b\ b = t$
⟨*proof*⟩

lemma $notin\_alphabet_F\_imp\_splitLeaf_F\_id$[*simp*]:
$a \notin alphabet_F\ ts \Longrightarrow splitLeaf_F\ ts\ w_a\ a\ w_b\ b = ts$
⟨*proof*⟩

lemma *alphabet_splitLeaf*[*simp*]:
alphabet (*splitLeaf* t w_a a w_b b) =
(*if* $a \in alphabet\ t$ *then* $alphabet\ t \cup \{b\}$ *else* $alphabet\ t$)
⟨*proof*⟩

lemma *consistent_splitLeaf*[*simp*]:
⟦*consistent* t; $b \notin alphabet\ t$⟧ ⟹ *consistent* (*splitLeaf* t w_a a w_b b)

⟨*proof*⟩

lemma *freq_splitLeaf*[*simp*]:
⟦*consistent t*; *b* ∉ *alphabet t*⟧ ⟹
freq (*splitLeaf t* w_a *a* w_b *b*) =
(*if a* ∈ *alphabet t then* (λ*c*. *if c* = *a then* w_a *else if c* = *b then* w_b *else freq t c*)
else freq t)
⟨*proof*⟩

lemma *weight_splitLeaf*[*simp*]:
⟦*consistent t*; *a* ∈ *alphabet t*; *freq t a* = $w_a + w_b$⟧ ⟹
weight (*splitLeaf t* w_a *a* w_b *b*) = *weight t*
⟨*proof*⟩

lemma *cost_splitLeaf*[*simp*]:
⟦*consistent t*; *a* ∈ *alphabet t*; *freq t a* = $w_a + w_b$⟧ ⟹
cost (*splitLeaf t* w_a *a* w_b *b*) = *cost t* + w_a + w_b
⟨*proof*⟩

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 ($t_1 \cdot t_2 \cdot ts$) =
(*weight* t_1 ≤ *weight* t_2 ∧ *sortedByWeight* ($t_2 \cdot ts$))

The function obeys the following fairly obvious laws.

lemma *sortedByWeight_Cons_imp_sortedByWeight*:
sortedByWeight ($t \cdot ts$) ⟹ *sortedByWeight ts*
⟨*proof*⟩

lemma *sortedByWeight_Cons_imp_forall_weight_ge*:
sortedByWeight ($t \cdot ts$) ⟹ ∀*u* ∈ *set ts*. *weight u* ≥ *weight t*
⟨*proof*⟩

lemma *sortedByWeight_insortTree*:
⟦*sortedByWeight ts*; *height t* = 0; $height_F$ *ts* = 0⟧ ⟹
sortedByWeight (*insortTree t ts*)
⟨*proof*⟩

4.8 Pair of Minimal Symbols

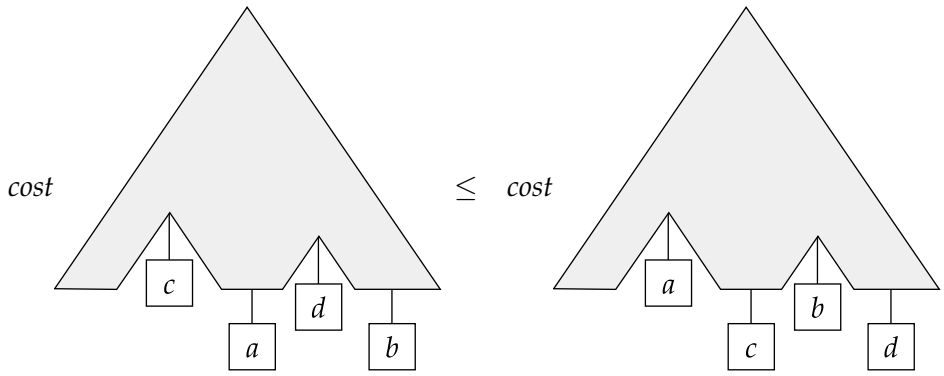
The *minima* predicate expresses that two symbols $a, b \in$ *alphabet t* have the lowest frequencies in the tree t. Minimal symbols need not be uniquely defined.

definition *minima* :: *'a tree* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
minima t a b =
(*a* ∈ *alphabet t* ∧ *b* ∈ *alphabet t* ∧ *a* ≠ *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 does not increase the cost. Graphically, we have



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:

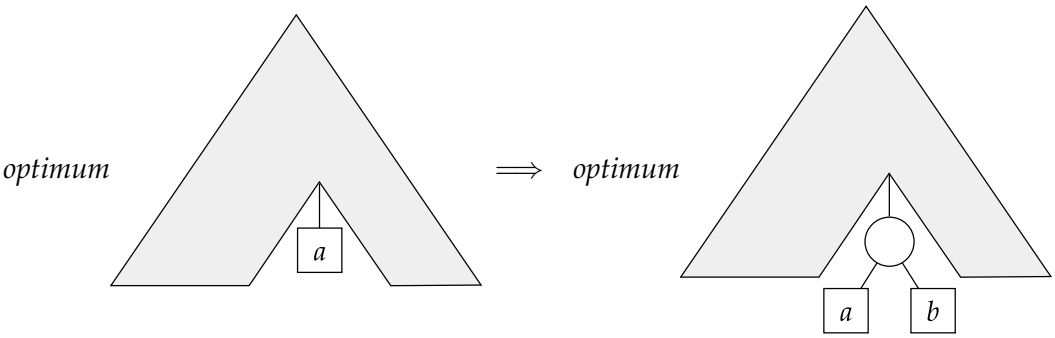
> Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

lemma *cost_swapFourSyms_le*:

assumes
consistent t minima t a b c ∈ *alphabet t d* ∈ *alphabet t*
depth t c = *height t depth t d* = *height t c* ≠ *d*
shows *cost* (*swapFourSyms t a b c d*) ≤ *cost t*
⟨*proof*⟩

5.2 Leaf Split Optimality Lemma

The tree *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 *freq t a* = $w_a + w_b$. Graphically:



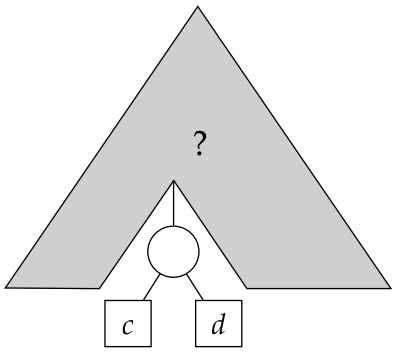
This corresponds to the following fragment of Knuth's proof:

> Now it is easy to prove that the weighted path length of such a tree is minimized if and only if the tree with
>
> 
>
>
> has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$.

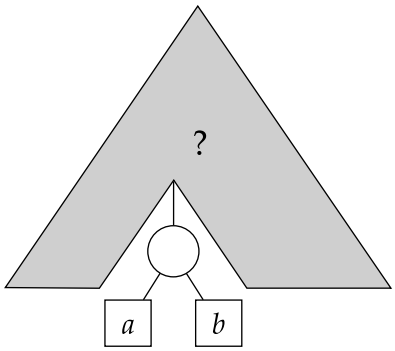
We only need the "if" direction of Knuth's equivalence. Lemma 16.3 in Cormen et al. [4, p. 391] expresses essentially the same property:

> 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.

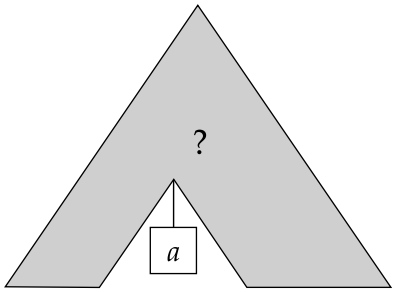
The proof is as follows: We assume that t has a cost less than or equal to that of any other comparable tree v and show that *splitLeaf* t w_a a w_b b has a cost less than or equal to that of any other comparable tree u. By *exists_at_height* and *depth_height_imp_sibling_ne*, we know that some symbols c and d appear in sibling nodes at the very bottom of u:



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



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:

$$\begin{array}{ll}
& \mathit{cost}\ (\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b) \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{cost_splitLeaf}) \\
& \mathit{cost}\ t + w_a + w_b \\
\leq & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{assumption}) \\
& \mathit{cost}\ (\overbrace{\mathit{mergeSibling}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d)\ a}^{v}) + w_a + w_b \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{cost_mergeSibling}) \\
& \mathit{cost}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d) \\
\leq & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{cost_swapFourSyms_le}) \\
& \mathit{cost}\ u.
\end{array}$$

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 $\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b$ and show that there exists a tree v with a lower cost than t, contradicting the hypothesis that t is optimum. In place of *cost_swapFourSyms_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*:
$[\![\forall c \in \mathit{alphabet}\ t.\ w_a \leq \mathit{freq}\ t\ c \wedge w_b \leq \mathit{freq}\ t\ c;$
$\mathit{alphabet}\ u = \mathit{alphabet}\ t \cup \{b\};\ a \in \mathit{alphabet}\ u;\ a \neq b;$
$\mathit{freq}\ u = (\lambda c.\ \mathit{if}\ c = a\ \mathit{then}\ w_a\ \mathit{else\ if}\ c = b\ \mathit{then}\ w_b\ \mathit{else}\ \mathit{freq}\ t\ c)]\!] \Longrightarrow$
$\mathit{minima}\ u\ a\ b$
⟨*proof*⟩

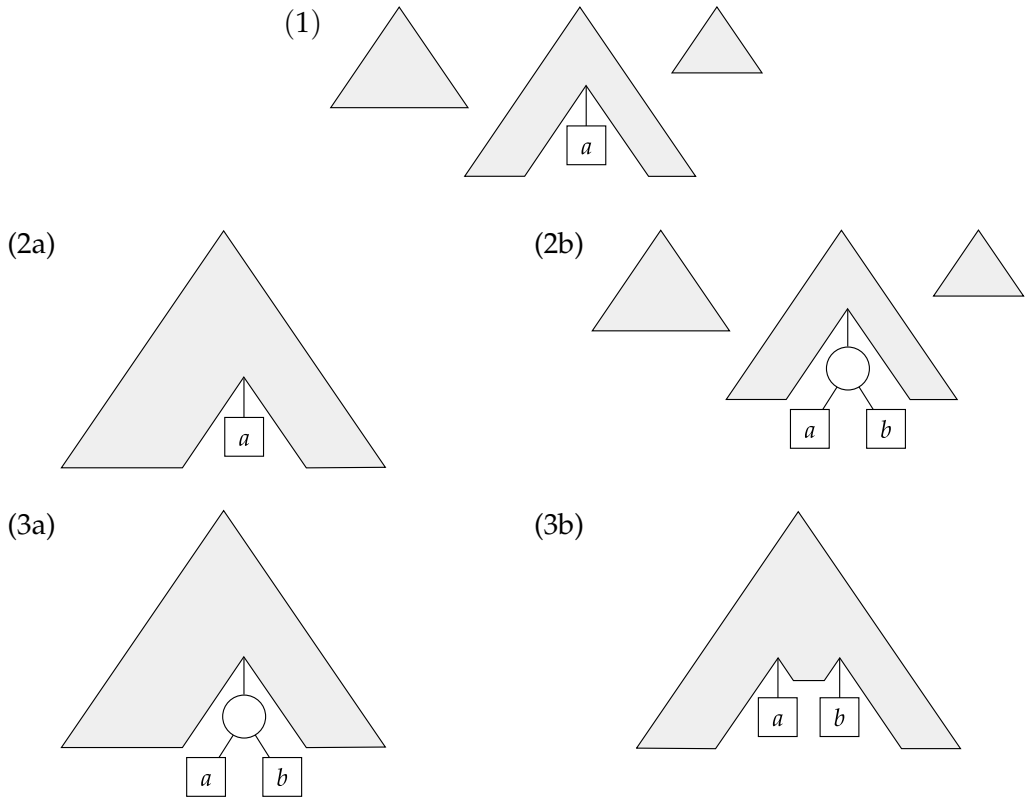
Now comes the key lemma.

lemma *optimum_splitLeaf*:
assumes $\mathit{consistent}\ t\ \mathit{optimum}\ t\ a \in \mathit{alphabet}\ t\ b \notin \mathit{alphabet}\ t$
$\mathit{freq}\ t\ a = w_a + w_b\ \forall c \in \mathit{alphabet}\ t.\ \mathit{freq}\ t\ c \geq w_a \wedge \mathit{freq}\ t\ c \geq w_b$
shows $\mathit{optimum}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b)$
⟨*proof*⟩

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 does not 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:

[3] Thomas Cormen commented that this step will be clarified in the next edition of the book.



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

$$splitLeaf\ (huffman\ ts)\ w_a\ a\ w_b\ b = huffman\ (splitLeaf_F\ ts\ w_a\ a\ w_b\ b)$$

when ts is consistent, $a \in alphabet_F\ ts$, $b \notin alphabet_F\ ts$, and $freq_F\ ts\ a = w_a + w_b$. But before we can prove this commutativity lemma, we need to introduce a few simple lemmas.

lemma *cachedWeight_splitLeaf*[*simp*]:
$cachedWeight\ (splitLeaf\ t\ w_a\ a\ w_b\ b) = cachedWeight\ t$
⟨*proof*⟩

lemma $splitLeaf_F$*_insortTree_when_in_alphabet_left*[*simp*]:
$[\![a \in alphabet\ t;\ consistent\ t;\ a \notin alphabet_F\ ts;\ freq\ t\ a = w_a + w_b]\!] \Longrightarrow$
$splitLeaf_F\ (insortTree\ t\ ts)\ w_a\ a\ w_b\ b = insortTree\ (splitLeaf\ t\ w_a\ a\ w_b\ b)\ ts$
⟨*proof*⟩

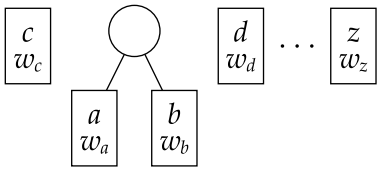
lemma $splitLeaf_F$*_insortTree_when_in_alphabet*$_F$*_tail*[*simp*]:
$[\![a \in alphabet_F\ ts;\ consistent_F\ ts;\ a \notin alphabet\ t;\ freq_F\ ts\ a = w_a + w_b]\!] \Longrightarrow$
$splitLeaf_F\ (insortTree\ t\ ts)\ w_a\ a\ w_b\ b =$

insortTree t (*splitLeaf*$_F$ *ts* w_a *a* w_b *b*)
⟨*proof*⟩

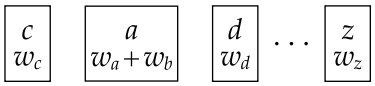
We are now ready to prove the commutativity lemma.

lemma *splitLeaf_huffman_commute*:
⟦*consistent*$_F$ *ts*; *a* ∈ *alphabet*$_F$ *ts*; *freq*$_F$ *ts a* = w_a + w_b⟧ ⟹
splitLeaf (*huffman ts*) w_a *a* w_b *b* = *huffman* (*splitLeaf*$_F$ *ts* w_a *a* w_b *b*)
⟨*proof*⟩

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



gives the same result as applying the algorithm on the "flat" forest



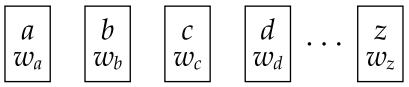
followed by splitting the leaf node *a* into two nodes *a*, *b* with frequencies w_a, w_b. The lemma effectively provides a way to flatten the forest at each step of the algorithm. Its invocation is implicit in the textbook proof.

5.4 Optimality Theorem

We are one lemma away from our main result.

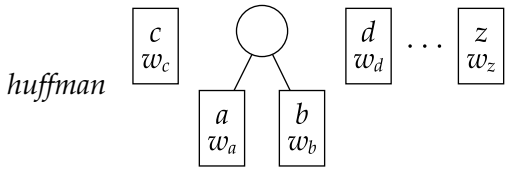
theorem *optimum_huffman*:
⟦*consistent*$_F$ *ts*; *height*$_F$ *ts* = 0; *sortedByWeight ts*; *ts* ≠ []⟧ ⟹
optimum (*huffman ts*)

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

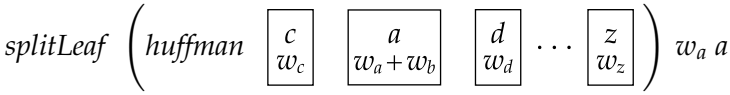


with $w_a \leq w_b \leq w_c \leq w_d \leq \cdots \leq w_z$. If *ts* consists of a single leaf node, the node has cost 0 and is therefore optimum. If *ts* has length 2 or more, the first step of

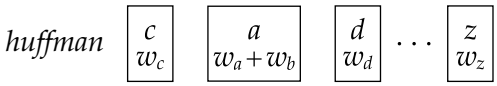
the algorithm leaves us with the term



In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By *splitLeaf_huffman_commute*, the above tree equals



To prove that this tree is optimum, it suffices by *optimum_splitLeaf* to show that



is optimum, which follows from the induction hypothesis.

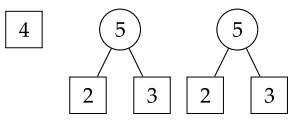
⟨*proof*⟩
end

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.

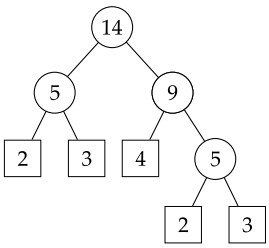
As a side note, the *optimum_huffman* theorem assumes that the forest *ts* passed to *huffman* consists exclusively of leaf nodes. It is tempting to relax this restriction, by requiring instead that the forest *ts* has the lowest cost among forests of the same size. We would define optimality of a forest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

$$\begin{aligned} \mathit{optimum}_F\ \mathit{ts}\ =\ (\forall \mathit{us}.\ & \mathit{length}\ \mathit{ts} = \mathit{length}\ \mathit{us} \longrightarrow \mathit{consistent}_F\ \mathit{us} \longrightarrow \\ & \mathit{alphabet}_F\ \mathit{ts} = \mathit{alphabet}_F\ \mathit{us} \longrightarrow \mathit{freq}_F\ \mathit{ts} = \mathit{freq}_F\ \mathit{us} \longrightarrow \\ & \mathit{cost}_F\ \mathit{ts} \leq \mathit{cost}_F\ \mathit{us}) \end{aligned}$$

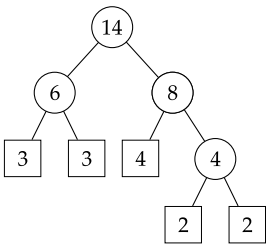
with $\mathit{cost}_F\ [] = 0$ and $\mathit{cost}_F\ (t \cdot \mathit{ts}) = \mathit{cost}\ t + \mathit{cost}_F\ \mathit{ts}$. However, the modified proposition does not hold. A counterexample is the optimum forest



for which the algorithm constructs the tree



of greater cost than



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

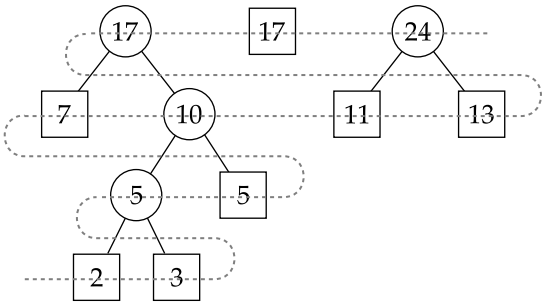
The goal of most formal proofs is to increase our confidence in a result. In the case of Huffman's algorithm, however, the chances that a bug would have gone unnoticed for the 56 years since its publication, under the scrutiny of leading computer scientists, seem extremely low; and the existence of a Coq proof should be sufficient to remove any remaining doubts.

The main contribution of this document 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 did not 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:



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

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] Stephan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, 230–239, IEEE Computer Society, 2004. Available online at `http://isabelle.in.tum.de/~nipkow/pubs/sefm04.html`.

[3] Lukas Bulwahn and Alexander Krauss. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, Volume 4732 of Lecture Notes in Computer Science, 38–53, Springer-Verlag, 2007. Available online at http://www4.in.tum.de/~krauss/lexord/lexord.pdf.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms* (Second Edition). MIT Press, 2001 and McGraw-Hill, 2002.

[5] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[6] Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, Volume 4732 of Lecture Notes in Computer Science, 128–143, Springer-Verlag, 2007. Available online at http://es.cs.uni-kl.de/TPHOLs-2007/proceedings/B-128.pdf.

[7] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers* **40**(9):1098–1101, September 1952. Available online at http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf.

[8] Donald E. Knuth. *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms* (Third Edition). Addison-Wesley, 1997.

[9] Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. Department of Informatics, Technische Universität München, 2007. Updated version, http://isabelle.in.tum.de/doc/functions.pdf, June 8, 2008.

[10] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Ph.D. thesis, Department of Informatics, Technische Universität München, 2009.

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML* (Revised Edition). MIT Press, 1997.

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Volume 2283 of Lecture Notes in Computer Science, Springer-Verlag, 2002. Updated version, http://isabelle.in.tum.de/doc/tutorial.pdf, June 8, 2008.

[13] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development* **20**(3):198–203, May 1976. Available online at http://www.research.ibm.com/journal/rd/203/ibmrd2003B.pdf.

[14] Laurent Théry. *A Correctness Proof of Huffman Algorithm*. http://coq.inria.fr/contribs/Huffman.html, October 2003.

[15] Laurent Théry. *Formalising Huffman's Algorithm*. Technical report TRCS 034/2004, Department of Informatics, University of L'Aquila, 2004.

[16] Markus Wenzel. *The Isabelle/Isar Reference Manual*. Department of Informatics, Technische Universität München, 2002. Updated version, http://isabelle.in.tum.de/doc/isar-ref.pdf, June 8, 2008.