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

October 15, 2008

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 codeword, 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 some bits by using shorter codewords 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 codeword for a. Information theory tells us that a code is optimum if for each source symbol c the codeword 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 [10], 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 codewords 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

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

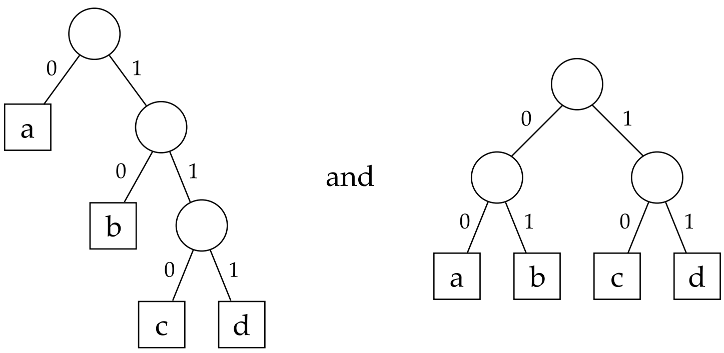
Encoding 'abacabad' produces the 14-bit codeword 01001100100111. The code C_1 is optimum: No code that unambiguously encodes source symbols one at a time could do better than C_1 on the input 'abacabad'. In particular, with a fixed-length code such as

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

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

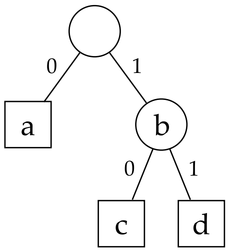
1.2 Binary Trees

Inside a program, binary codes like C_1 and C_2 can be represented by binary trees. For example, the trees



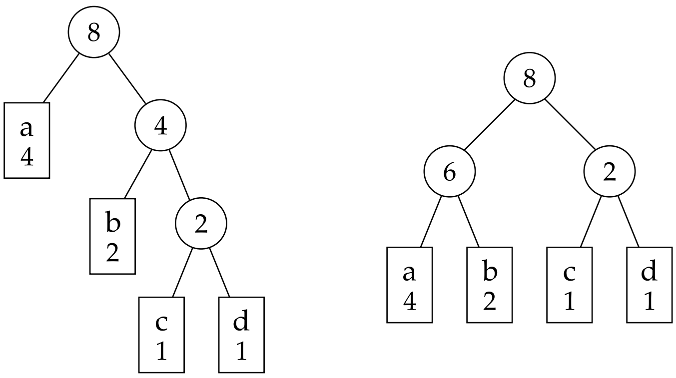
correspond to C_1 and C_2, respectively. The codeword for a symbol is given along the path from the root to that symbol, with 0 meaning "left child" and 1 meaning "right child".

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no codeword 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 doesn't have this property, consider the code associated with the tree



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

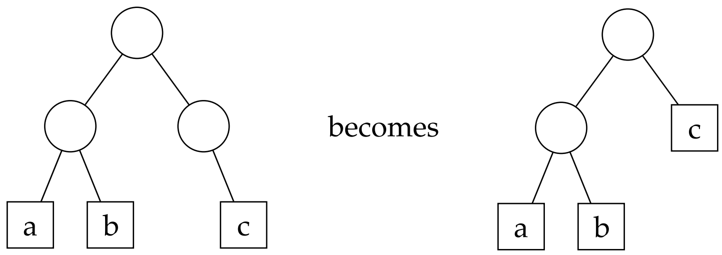
Each node in a code tree is assigned a *weight*. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. Code trees can be annotated with their weights:



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

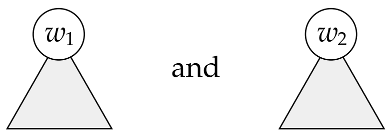
[1] Strictly speaking, there is another potential source of ambiguity. If the alphabet consists of a single symbol a, that symbol could be mapped to the empty codeword, and then any string $aa \ldots a$ would map to the empty bit sequence, giving the decoder no way to recover the original string's length.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

child can advantageously be eliminated; for example,

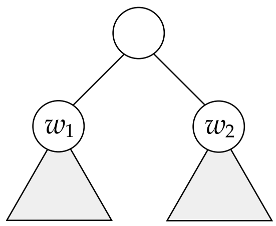


1.3 Huffman's Algorithm

David Huffman [5] discovered a simple algorithm for constructing an optimum prefix code tree for specified symbol frequencies: Create a forest consisting of only leaf nodes, one for each symbol in the alphabet. Then take the two trees



with the lowest weights and replace them with the tree

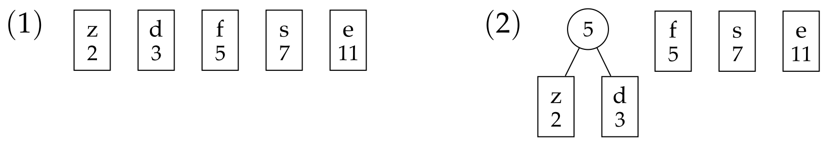


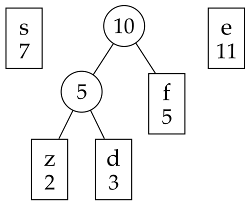
Repeat this process until only one tree is left.

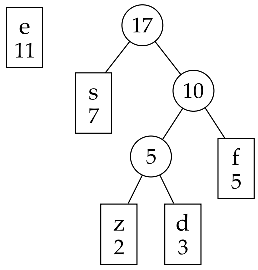
As an illustration, executing the algorithm for the frequencies

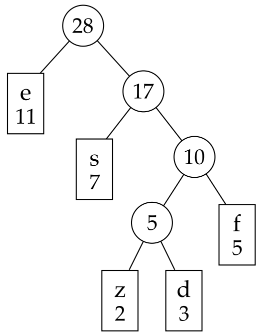
$$f_{\mathrm{d}} = 3,\ \ f_{\mathrm{e}} = 11,\ \ f_{\mathrm{f}} = 5,\ \ f_{\mathrm{s}} = 7,\ \ f_{\mathrm{z}} = 2$$

gives rise to the following sequence of states:



(3)


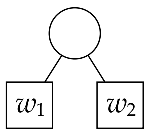
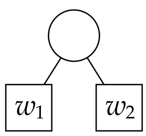
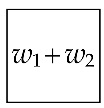
(4)


(5)


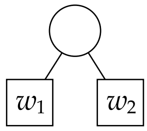
Tree (5) is an optimum tree for the given frequencies.

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 [6, 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
>
> 
>
>
> replaced by
>
> 
>
>
> 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



(The formalization works without it.)

Cormen et al. [3, 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 [9]. 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 [11, 12], which through its "cover" concept represents a considerable departure from the standard 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 [13] 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 [6, 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 a fragment of higher-order logic [4, 9]. The metalogical operators are material implication, written $[\![\varphi_1;\ \ldots;\ \varphi_n]\!] \Longrightarrow \psi$ ("if φ_1 and ... and φ_n, then ψ"), universal quantification, written $\bigwedge x_1 \ldots x_n.\ \psi$ ("for all $x_1, \ldots, x_n$ we have ψ"), 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 ($\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\forall$, and $\exists$) on terms of type *bool*. In addition, $=$ expresses equivalence. The formulas $\bigwedge x_1 \ldots x_m.\ [\![\varphi_1;\ \ldots;\ \varphi_n]\!] \Longrightarrow \psi$ and $\forall x_1. \ldots \forall x_m.\ \varphi_1 \wedge \cdots \wedge \varphi_n \longrightarrow \psi$ are logically equivalent, but they interact differently with Isabelle's proof tactics.

The term language consists of simply typed λ-terms written in an ML-like syntax [8]. In particular, 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 τ is its type. The type of total functions from $'a$ to $'b$ is written $'a \Rightarrow 'b$. Variables may range over functions.

The type of natural numbers is called *nat*. The type of lists over type $'a$, written $'a$ *list*, features the empty list $[]$, the infix constructor $x \cdot xs$ (where x is an element of type $'a$ and xs is a list over $'a$), and the conversion function *set* from lists to sets. The type of sets over $'a$ is written $'a$ *set*. Operations on sets are written using traditional mathematical notations.

1.7 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*]
*min_max.sup_absorb*1 [*simp*]
*min_max.sup_absorb*2 [*simp*]

2 Definition of Prefix Code Trees and Forests

2.1 Tree Datatype

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 *InnerNode* 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
InnerNode nat (*'a tree*) (*'a tree*)

2.2 Forest Datatype

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

types *'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* $\Rightarrow$ *'a set* **where**
alphabet (*Leaf* w a) = $\{a\}$
alphabet (*InnerNode* w t_1 t_2) = *alphabet* $t_1 \cup$ *alphabet* t_2

For set 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 is very useful when developing proofs top down (together with **sorry**), has limited support for inductive 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*]:
$finite\ (alphabet\ t)$
by (*induct t*) *auto*

lemma *exists_in_alphabet*:
$\exists a.\ a \in alphabet\ t$
by (*induct t*) *auto*

2.4 Consistency

A tree is *consistent* if for each inner node the alphabet of the two subtrees are disjoint. Intuitively, this means that every symbol in the alphabet occurs in exactly one leaf node. Although this well-formedness property isn't mentioned in algorithms textbooks [1, 3, 6], it is essential and appears as an assumption in many of our lemmas.

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

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

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

CONSISTENCY CHECK: The property holds if t is inconsistent. (This is always the case when *consistent t* occurs among the property's assumptions.)

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

INDUCTION STEP: $t = InnerNode\ w\ t_1\ t_2$.

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 check, 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*:
$\llbracket\bigwedge t\ a.\ \neg\ consistent\ t \Longrightarrow P\ t\ a;$
$\bigwedge w_b\ b\ a.\ P\ (Leaf\ w_b\ b)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$\quad\llbracket consistent\ t_1;\ consistent\ t_2;\ alphabet\ t_1 \cap alphabet\ t_2 = \{\};$
$\quad a \in alphabet\ t_1;\ a \notin alphabet\ t_2;\ P\ t_1\ a;\ P\ t_2\ a\rrbracket \Longrightarrow$
$\quad P\ (InnerNode\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$\quad\llbracket consistent\ t_1;\ consistent\ t_2;\ alphabet\ t_1 \cap alphabet\ t_2 = \{\};$
$\quad a \notin alphabet\ t_1;\ a \in alphabet\ t_2;\ P\ t_1\ a;\ P\ t_2\ a\rrbracket \Longrightarrow$
$\quad P\ (InnerNode\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$\quad\llbracket consistent\ t_1;\ consistent\ t_2;\ alphabet\ t_1 \cap alphabet\ t_2 = \{\};$
$\quad a \notin alphabet\ t_1;\ a \notin alphabet\ t_2;\ P\ t_1\ a;\ P\ t_2\ a\rrbracket \Longrightarrow$
$\quad P\ (InnerNode\ w\ t_1\ t_2)\ a\rrbracket \Longrightarrow$
$P\ t\ a$

The proof relies on the **induct_scheme** and **lexicographic_order** tactics, which automate the most tedious aspects of deriving induction rules. An alternative would have been to perform a standard structural induction on t and proceed by cases.

apply *induct_scheme*
  apply *atomize_elim*
  apply (*rename_tac p*)
  apply (*case_tac p, simp*)
  apply *clarify*
  apply (*rename_tac t a*)
  apply (*case_tac t*)
   apply *fastsimp*
  apply *fastsimp*

by *lexicographic_order*

2.5 Symbol Depths

The *depth* of a symbol is the length of the path from the root to the leaf node labeled with that 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.

primrec $depth :: 'a\ tree \Rightarrow 'a \Rightarrow nat$ **where**
$depth\ (Leaf\ w\ b)\ a = 0$
$depth\ (InnerNode\ w\ t_1\ t_2)\ a =$
  $(if\ a \in alphabet\ t_1\ then\ depth\ t_1\ a + 1$
  $else\ if\ a \in alphabet\ t_2\ then\ depth\ t_2\ a + 1$
  $else\ 0)$

2.6 Height

The *height* of a tree is the length of the longest path from the root to a leaf node. 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 $height :: 'a\ tree \Rightarrow nat$ **where**
$height\ (Leaf\ w\ a) = 0$
$height\ (InnerNode\ w\ t_1\ t_2) = max\ (height\ t_1)\ (height\ t_2) + 1$

primrec $height_F :: 'a\ forest \Rightarrow nat$ **where**
$height_F\ [] = 0$
$height_F\ (t \cdot ts) = max\ (height\ t)\ (height_F\ ts)$

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*:
$depth\ t\ a \le height\ t$
by (*induct t*) *auto*

lemma *exists_at_height*:
$consistent\ t \Longrightarrow \exists a \in alphabet\ t.\ depth\ t\ a = height\ t$
proof (*induct t*)
 case *Leaf* **thus** *case* **by** *simp*
next
 case $(InnerNode\ w\ t_1\ t_2)$
 note $hyps = InnerNode$

let $t = InnerNode\ w\ t_1\ t_2$
from *hyps* **obtain** *b* **where** *b*: $b \in alphabet\ t_1$ $depth\ t_1\ b = height\ t_1$ **by** *auto*
from *hyps* **obtain** *c* **where** *c*: $c \in alphabet\ t_2$ $depth\ t_2\ c = height\ t_2$ **by** *auto*
let $a = if\ height\ t_1 \geq height\ t_2\ then\ b\ else\ c$
from *b c* **have** $a \in alphabet\ t$ $depth\ t\ a = height\ t$
using ⟨*consistent t*⟩ **by** *auto*
thus $\exists a \in alphabet\ t.\ depth\ t\ a = height\ t$ **..**
qed

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
by (*cut_tac* $t = t_1$ **and** $a = a$ **in** *depth_le_height*) *simp*

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
by (*cut_tac* $t = t_2$ **and** $a = a$ **in** *depth_le_height*) *simp*

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 (*cases t*)
case *Leaf* **thus** *thesis* **using** *assms* **by** *simp*
next
case ($InnerNode\ w\ t_1\ t_2$)
note $t = InnerNode$
from *exists_in_alphabet* **obtain** *b* **where** *b*: $b \in alphabet\ t_1$ **..**
from *exists_in_alphabet* **obtain** *c* **where** *c*: $c \in alphabet\ t_2$ **..**
from *b c* **have** *bc*: $b \neq c$ **using** *t* ⟨*consistent t*⟩ **by** *fastsimp*
show *thesis*
proof (*cases u*)
case *Leaf* **thus** *thesis* **using** *b c bc t assms* **by** *auto*
next
case *InnerNode* **thus** *thesis* **by** *simp*
qed
qed

2.7 Symbol Frequencies

The *frequency* of a symbol 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 generalization to forests is straightforward. If two trees have the same alphabet and symbol frequencies, we say that they are *compatible*.

primrec *freq* :: $'a\ tree \Rightarrow 'a \Rightarrow nat$ **where**
$freq\ (Leaf\ w\ a) = (\lambda b.\ if\ b = a\ then\ w\ else\ 0)$
$freq\ (InnerNode\ w\ t_1\ t_2) = (\lambda b.\ freq\ t_1\ b + freq\ t_2\ b)$

primrec $freq_F$ :: $'a\ forest \Rightarrow 'a \Rightarrow nat$ **where**
$freq_F\ [] = (\lambda b.\ 0)$
$freq_F\ (t \cdot ts) = (\lambda b.\ freq\ t\ b + freq_F\ ts\ b)$

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*]:
$a \notin alphabet\ t \Longrightarrow freq\ t\ a = 0$
by (*induct t*) *simp+*

lemma $notin\_alphabet_F\_imp\_freq_F\_0$ [*simp*]:
$a \notin alphabet_F\ ts \Longrightarrow freq_F\ ts\ a = 0$
by (*induct ts*) *simp+*

lemma *freq_0_right* [*simp*]:
$[\![alphabet\ t_1 \cap alphabet\ t_2 = \{\};\ a \in alphabet\ t_1]\!] \Longrightarrow freq\ t_2\ a = 0$
by (*auto intro*: *notin_alphabet_imp_freq_0 simp*: *disjoint_iff_not_equal*)

lemma *freq_0_left* [*simp*]:
$[\![alphabet\ t_1 \cap alphabet\ t_2 = \{\};\ a \in alphabet\ t_2]\!] \Longrightarrow freq\ t_1\ a = 0$
by (*auto simp*: *disjoint_iff_not_equal*)

We close this section with a more technical lemma.

lemma $height_F\_0\_imp\_Leaf\_freq_F\_in\_set$:
$[\![consistent_F\ ts;\ height_F\ ts = 0;\ a \in alphabet_F\ ts]\!] \Longrightarrow$
$Leaf\ (freq_F\ ts\ a)\ a \in set\ ts$
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 is more robust than relying on the cache and simplifies reasoning.

primrec $\mathit{weight} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$ **where**
$\mathit{weight}\ (\mathit{Leaf}\ w\ a) = w$
$\mathit{weight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2$

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

lemma *weight_eq_Sum_freq*:
$\mathit{consistent}\ t \Longrightarrow \mathit{weight}\ t = \sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a$
by $(\mathit{induct}\ t)$ $(\mathit{auto\ simp}\colon \mathit{setsum\_Un\_disjoint})$

The assumption *consistent t* is not necessary, but it simplifies the proof by letting us invoke the lemma *setsum_Un_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 $\sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a$. It obeys a simple recursive law.

primrec $\mathit{cost} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$ **where**
$\mathit{cost}\ (\mathit{Leaf}\ w\ a) = 0$
$\mathit{cost}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2$

One interpretation of this recursive law is that the cost of a tree is the sum of the weights of its inner nodes [6, p. 405]. (Recall that $\mathit{weight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2$.) 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*:
$\mathit{consistent}\ t \Longrightarrow \mathit{cost}\ t = \sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a$

The proof is by structural induction on t. If $t = \mathit{Leaf}\ w\ b$, both sides of the equation simplify to 0. This leaves the case $t = \mathit{InnerNode}\ w\ t_1\ t_2$. Let A, A_1, and A_2 stand for *alphabet* t, *alphabet* t_1, and *alphabet* t_2, respectively. We have

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

The structured proof closely follows this argument.

proof (*induct t*)
 case *Leaf* **thus** *case* **by** *simp*
next
 case (*InnerNode w* t_1 t_2)
 let $t = InnerNode\ w\ t_1\ t_2$
 let $A = alphabet\ t$ **and** $A_1 = alphabet\ t_1$ **and** $A_2 = alphabet\ t_2$
 note $c = \langle consistent\ t\rangle$
 note $hyps = InnerNode$
 have d_2: $\bigwedge a.\ [\![A_1 \cap A_2 = \{\};\ a \in A_2]\!] \Longrightarrow depth\ t\ a = depth\ t_2\ a + 1$
  by *auto*
 have $cost\ t = weight\ t_1 + cost\ t_1 + weight\ t_2 + cost\ t_2$ **by** *simp*
 also have $\ldots = weight\ t_1 + (\sum a \in A_1.\ freq\ t_1\ a \times depth\ t_1\ a)\ +$
   $weight\ t_2 + (\sum a \in A_2.\ freq\ t_2\ a \times depth\ t_2\ a)$
  using *hyps* **by** *simp*
 also have $\ldots = weight\ t_1 + (\sum a \in A_1.\ freq\ t_1\ a \times (depth\ t\ a - 1))\ +$
   $weight\ t_2 + (\sum a \in A_2.\ freq\ t_2\ a \times (depth\ t\ a - 1))$
  using c d_2 **by** *simp*
 also have $\ldots = weight\ t_1 + (\sum a \in A_1.\ freq\ t_1\ a \times depth\ t\ a)$

$-\ (\sum a \in A_1.\ \mathit{freq}\ t_1\ a)\ +$
$\mathit{weight}\ t_2 + (\sum a \in A_2.\ \mathit{freq}\ t_2\ a \times \mathit{depth}\ t\ a)$
$-\ (\sum a \in A_2.\ \mathit{freq}\ t_2\ a)$
using *c* d_2 **by** (*simp add*: *setsum_addf*)
also have ... $= (\sum a \in A_1.\ \mathit{freq}\ t_1\ a \times \mathit{depth}\ t\ a)\ +$
$(\sum a \in A_2.\ \mathit{freq}\ t_2\ a \times \mathit{depth}\ t\ a)$
using *c* **by** (*simp add*: *weight_eq_Sum_freq*)
also have ... $= (\sum a \in A_1.\ \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a)\ +$
$(\sum a \in A_2.\ \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a)$
using *c* **by** *auto*
also have ... $= (\sum a \in A_1 \cup A_2.\ \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a)$
using *c* **by** (*simp add*: *setsum_Un_disjoint*)
also have ... $= (\sum a \in A.\ \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a)$ **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*]:
$\mathit{height}\ t = 0 \Longrightarrow \mathit{cost}\ t = 0$
by (*case_tac t*) *simp+*

2.10 Optimality

A tree is optimum if and only if its cost is not greater than the cost of any compatible tree. We can ignore inconsistent trees without loss of generality.

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

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.

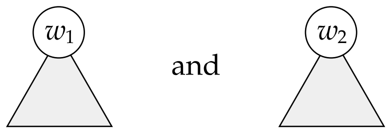
primrec *cachedWeight* :: *'a tree* ⇒ *nat* **where**
$\mathit{cachedWeight}\ (\mathit{Leaf}\ w\ a) = w$
$\mathit{cachedWeight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = w$

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

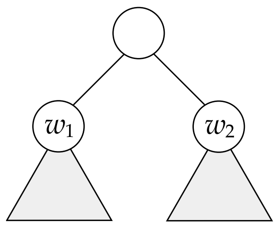
lemma *height_0_imp_cachedWeight_eq_weight* [*simp*]:
height t = 0 $\Longrightarrow$ *cachedWeight t* = *weight t*
by (*case_tac t*) *simp*+

3.2 Tree Union

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



and returns the tree



definition *uniteTrees* :: *'a tree* $\Rightarrow$ *'a tree* $\Rightarrow$ *'a tree* **where**
uniteTrees t_1 t_2 $\equiv$ *InnerNode* (*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 $\cup$ *alphabet* t_2
by (*simp add*: *uniteTrees_def*)

lemma *consistent_uniteTrees* [*simp*]:
$[\![$*consistent* t_1; *consistent* t_2; *alphabet* t_1 $\cap$ *alphabet* t_2 = {}$]\!]$ $\Longrightarrow$
consistent (*uniteTrees* t_1 t_2)
by (*simp add*: *uniteTrees_def*)

lemma *freq_uniteTrees* [*simp*]:
freq (*uniteTrees* t_1 t_2) = (λa. *freq* t_1 *a* + *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 *insortTree* :: $'a$ *tree* $\Rightarrow$ $'a$ *forest* $\Rightarrow$ $'a$ *forest* **where**
$insortTree\ u\ [] = [u]$
$insortTree\ u\ (t \cdot ts) =$
$\quad (if\ cachedWeight\ u \leq cachedWeight\ t\ then\ u \cdot t \cdot ts$
$\qquad\qquad\qquad\qquad else\ t \cdot insortTree\ u\ ts)$

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

lemma *length_insortTree* [*simp*]:
$length\ (insortTree\ t\ ts) = length\ ts + 1$
by (*induct ts*) *simp*+

lemma *insortTree_ne_Nil* [*simp*]:
$insortTree\ t\ ts \neq []$
by (*induct 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*]:
$alphabet_F\ (insortTree\ t\ ts) = alphabet\ t \cup alphabet_F\ ts$
by (*induct ts*) *auto*

lemma $consistent_F$*_insortTree* [*simp*]:
$consistent_F\ (insortTree\ t\ ts) = consistent_F\ (t \cdot ts)$
by (*induct ts*) *auto*

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

fun *huffman* :: $'a$ *forest* $\Rightarrow$ $'a$ *tree* **where**
$huffman\ [t] = t$
$huffman\ (t_1 \cdot t_2 \cdot ts) = huffman\ (insortTree\ (uniteTrees\ t_1\ t_2)\ ts)$

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

theorem *alphabet_huffman* [*simp*]:
$ts \neq [] \Longrightarrow alphabet\ (huffman\ ts) = alphabet_F\ ts$
by (*induct ts rule*: *huffman.induct*) *auto*

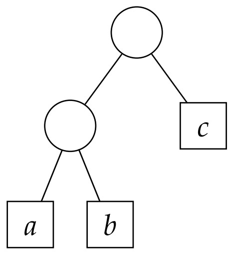
theorem *consistent_huffman* [*simp*]:
$[\![consistent_F\ ts;\ ts \neq []]\!] \Longrightarrow consistent\ (huffman\ ts)$
by (*induct ts rule*: *huffman.induct*) *simp+*

theorem *freq_huffman* [*simp*]:
$ts \neq [] \Longrightarrow freq\ (huffman\ ts) = freq_F\ ts$
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. Thus, we have $sibling\ t\ a = b$, $sibling\ t\ b = a$, and $sibling\ t\ c = c$ for the tree

$t =$ 

fun $sibling :: 'a\ tree \Rightarrow 'a \Rightarrow 'a$ **where**
$sibling\ (Leaf\ w_b\ b)\ a = a$
$sibling\ (InnerNode\ w\ (Leaf\ w_b\ b)\ (Leaf\ w_c\ c))\ a =$
$\quad (if\ a = b\ then\ c\ else\ if\ a = c\ then\ b\ else\ a)$
$sibling\ (InnerNode\ w\ t_1\ t_2)\ a =$
$\quad (if\ a \in alphabet\ t_1\ then\ sibling\ t_1\ a$
$\quad\ else\ if\ a \in alphabet\ t_2\ then\ sibling\ t_2\ a$
$\quad\ else\ a)$

Because *sibling* is defined using sequential pattern matching [7], reasoning about it can become tedious. Simplification rules therefore play an important role.

lemma *notin_alphabet_imp_sibling_id* [*simp*]:

$a \notin \mathit{alphabet}\ t \Longrightarrow \mathit{sibling}\ t\ a = a$
by $(\mathit{induct}\ t\ a\ \mathit{rule}: \mathit{sibling.induct})$ *simp+*

lemma *height_0_imp_sibling_id* [*simp*]:
$\mathit{height}\ t = 0 \Longrightarrow \mathit{sibling}\ t\ a = a$
by (*case_tac t*) *simp+*

lemma *height_gt_0_in_alphabet_imp_sibling_left* [*simp*]:
$[\![\mathit{height}\ t_1 > 0;\ a \in \mathit{alphabet}\ t_1]\!] \Longrightarrow$
$\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a = \mathit{sibling}\ t_1\ a$
by $(\mathit{case\_tac}\ t_1)$ *simp+*

lemma *height_gt_0_in_alphabet_imp_sibling_right* [*simp*]:
$[\![\mathit{height}\ t_2 > 0;\ a \in \mathit{alphabet}\ t_1]\!] \Longrightarrow$
$\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a = \mathit{sibling}\ t_1\ a$
by $(\mathit{case\_tac}\ t_2)$ *simp+*

lemma *height_gt_0_notin_alphabet_imp_sibling_left* [*simp*]:
$[\![\mathit{height}\ t_1 > 0;\ a \notin \mathit{alphabet}\ t_1]\!] \Longrightarrow$
$\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a = \mathit{sibling}\ t_2\ a$
by $(\mathit{case\_tac}\ t_1)$ *simp+*

lemma *height_gt_0_notin_alphabet_imp_sibling_right* [*simp*]:
$[\![\mathit{height}\ t_2 > 0;\ a \notin \mathit{alphabet}\ t_1]\!] \Longrightarrow$
$\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a = \mathit{sibling}\ t_2\ a$
by $(\mathit{case\_tac}\ t_2)$ *simp+*

lemma *either_height_gt_0_imp_sibling* [*simp*]:
$\mathit{height}\ t_1 > 0 \lor \mathit{height}\ t_2 > 0 \Longrightarrow$
$\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a =$
$\quad (\mathit{if}\ a \in \mathit{alphabet}\ t_1\ \mathit{then}\ \mathit{sibling}\ t_1\ a\ \mathit{else}\ \mathit{sibling}\ t_2\ a)$
by *auto*

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

lemma *in_alphabet_imp_sibling_in_alphabet*:
$a \in \mathit{alphabet}\ t \Longrightarrow \mathit{sibling}\ t\ a \in \mathit{alphabet}\ t$
by $(\mathit{induct}\ t\ a\ \mathit{rule}: \mathit{sibling.induct})$ *auto*

lemma *sibling_ne_imp_sibling_in_alphabet*:
$\mathit{sibling}\ t\ a \neq a \Longrightarrow \mathit{sibling}\ t\ a \in \mathit{alphabet}\ t$
by (*metis notin_alphabet_imp_sibling_id in_alphabet_imp_sibling_in_alphabet*)

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

> BASE CASE: $t = \mathit{Leaf}\ w\ b$.
>
> INDUCTION STEP 1: $t = \mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c)$.

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

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

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 *InnerNode* 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.

CONSISTENCY CHECK: The property holds if t is inconsistent.

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

INDUCTION STEP 1: $t = \mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c)$ with $b \neq c$.

INDUCTION STEP 2: $t = \mathit{InnerNode}\ w\ t_1\ t_2$ 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*:
$[\![\bigwedge t\ a.\ \neg\ \mathit{consistent}\ t \Longrightarrow P\ t\ a;$
$\bigwedge w\ b\ a.\ P\ (\mathit{Leaf}\ w\ b)\ a;$
$\bigwedge w\ w_b\ b\ w_c\ c\ a.\ b \neq c \Longrightarrow P\ (\mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c))\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$\quad [\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};$
$\quad \mathit{height}\ t_1 > 0 \vee \mathit{height}\ t_2 > 0;\ a \in \mathit{alphabet}\ t_1;$
$\quad \mathit{sibling}\ t_1\ a \in \mathit{alphabet}\ t_1;\ a \notin \mathit{alphabet}\ t_2;$
$\quad \mathit{sibling}\ t_1\ a \notin \mathit{alphabet}\ t_2;\ P\ t_1\ a]\!] \Longrightarrow$
$\quad P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.$
$\quad [\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\};$
$\quad \mathit{height}\ t_1 > 0 \vee \mathit{height}\ t_2 > 0;\ a \notin \mathit{alphabet}\ t_1;$
$\quad \mathit{sibling}\ t_2\ a \notin \mathit{alphabet}\ t_1;\ a \in \mathit{alphabet}\ t_2;$
$\quad \mathit{sibling}\ t_2\ a \in \mathit{alphabet}\ t_2;\ P\ t_2\ a]\!] \Longrightarrow$
$\quad P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a;$

$\bigwedge w\ t_1\ t_2\ a.$
$[\![consistent\ t_1;\ consistent\ t_2;\ alphabet\ t_1 \cap alphabet\ t_2 = \{\};$
$height\ t_1 > 0 \vee height\ t_2 > 0;\ a \notin alphabet\ t_1;\ a \notin alphabet\ t_2]\!] \Longrightarrow$
$P\ (InnerNode\ w\ t_1\ t_2)\ a]\!] \Longrightarrow$
$P\ t\ a$
apply *induct_scheme*
apply *atomize_elim*
apply (*rename_tac p*)
apply (*case_tac p, simp*)
apply (*case_tac a, simp*)
apply *clarsimp*
apply (*rename_tac a* t_1 t_2)
apply (*case_tac height* $t_1 = 0 \wedge$ *height* $t_2 = 0$)
apply *simp*
apply (*case_tac* t_1)
apply (*case_tac* t_2)
apply *fastsimp*
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 $\Longrightarrow$ *sibling t* (*sibling t a*) = *a*
by (*induct t a rule*: *sibling_induct_consistent*) *simp+*

lemma *sibling_reciprocal*:
$[\![$*consistent t*; *sibling t a* = *b*$]\!] \Longrightarrow$ *sibling t b* = *a*
by *auto*

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

lemma *depth_sibling* [*simp*]:
consistent t $\Longrightarrow$ *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.

Notice 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**
$swapLeaves\ (Leaf\ w_c\ c)\ w_a\ a\ w_b\ b =$
$\quad (if\ c = a\ then\ Leaf\ w_b\ b\ else\ if\ c = b\ then\ Leaf\ w_a\ a\ else\ Leaf\ w_c\ c)$
$swapLeaves\ (InnerNode\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b =$
$\quad InnerNode\ w\ (swapLeaves\ t_1\ w_a\ a\ w_b\ b)\ (swapLeaves\ t_2\ w_a\ a\ w_b\ b)$

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*]:
$a \notin alphabet\ t \Longrightarrow swapLeaves\ t\ w\ a\ w'\ a = t$
by (*induct t*) *simp*+

lemma *swapLeaves_id* [*simp*]:
$consistent\ t \Longrightarrow swapLeaves\ t\ (freq\ t\ a)\ a\ (freq\ t\ a)\ a = t$
by (*induct t a rule*: *tree_induct_consistent*) *simp*+

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

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

lemma *consistent_swapLeaves* [*simp*]:
$consistent\ t \Longrightarrow consistent\ (swapLeaves\ t\ w_a\ a\ w_b\ b)$
by (*induct t*) (*auto simp*: *alphabet_swapLeaves*)

lemma *depth_swapLeaves_neither* [*simp*]:
$[\![consistent\ t;\ c \neq a;\ c \neq b]\!] \Longrightarrow depth\ (swapLeaves\ t\ w_a\ a\ w_b\ b)\ c = depth\ t\ c$
by (*induct t a rule*: *tree_induct_consistent*) (*auto simp*: *alphabet_swapLeaves*)

lemma *height_swapLeaves* [*simp*]:
$height\ (swapLeaves\ t\ w_a\ a\ w_b\ b) = height\ t$
by (*induct t*) *simp*+

lemma *freq_swapLeaves* [*simp*]:

$[\![consistent\ t;\ a \neq b]\!] \Longrightarrow$
$freq\ (swapLeaves\ t\ w_a\ a\ w_b\ b) =$
$\quad(\lambda c.\ if\ c = a\ then\ if\ b \in alphabet\ t\ then\ w_a\ else\ 0$
$\qquad else\ if\ c = b\ then\ if\ a \in alphabet\ t\ then\ w_b\ else\ 0$
$\qquad else\ freq\ t\ c)$
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

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

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 $freq\ t\ a \leq weight\ t$ (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*:
$[\![consistent\ t;\ a \neq b]\!] \Longrightarrow$
$if\ a \in alphabet\ t\ then$
$\ \ if\ b \in alphabet\ t\ then$
$\quad weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ a + freq\ t\ b =$
$\qquad weight\ t + w_a + w_b$
$\ \ else$
$\quad weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ a = weight\ t + w_b$
$else$
$\ \ if\ b \in alphabet\ t\ then$
$\quad weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ b = weight\ t + w_a$
$\ \ else$
$\quad weight\ (swapLeaves\ t\ w_a\ a\ w_b\ b) = weight\ t$
proof (*induct t a rule*: *tree_induct_consistent*)
— CONSISTENCY CHECK
case 1 **thus** *case* **by** *simp*
next
— BASE CASE: $t = Leaf\ w\ b$
case 2 **thus** *case* **by** *clarsimp*
next
— INDUCTION STEP: $t = InnerNode\ w\ t_1\ t_2$

— SUBCASE 1: a belongs to t_1 but not to t_2
case $(3\ w\ t_1\ t_2\ a)$ **show** *case*
proof *cases*
assume $b \in alphabet\ t_1$
moreover hence $b \notin alphabet\ t_2$ **using** 3 **by** *auto*
ultimately show *case* **using** 3 **by** *simp*
next
assume $b \notin alphabet\ t_1$ **thus** *case* **using** 3 **by** *auto*
qed
next
— SUBCASE 2: a belongs to t_2 but not to t_1
case $(4\ w\ t_1\ t_2\ a)$ **show** *case*
proof *cases*
assume $b \in alphabet\ t_1$
moreover hence $b \notin alphabet\ t_2$ **using** 4 **by** *auto*
ultimately show *case* **using** 4 **by** *simp*
next
assume $b \notin alphabet\ t_1$ **thus** *case* **using** 4 **by** *auto*
qed
next
— SUBCASE 3: a belongs to neither t_1 nor t_2
case $(5\ w\ t_1\ t_2\ a)$ **show** *case*
proof *cases*
assume $b \in alphabet\ t_1$
moreover hence $b \notin alphabet\ t_2$ **using** 5 **by** *auto*
ultimately show *case* **using** 5 **by** *simp*
next
assume $b \notin alphabet\ t_1$ **thus** *case* **using** 5 **by** *auto*
qed
qed

lemma *cost_swapLeaves*:
$[\![consistent\ t;\ a \neq b]\!] \Longrightarrow$
if $a \in alphabet\ t$ *then*
if $b \in alphabet\ t$ *then*
$cost\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ a \times depth\ t\ a$
$+ freq\ t\ b \times depth\ t\ b =$
$cost\ t + w_a \times depth\ t\ b + w_b \times depth\ t\ a$
else
$cost\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ a \times depth\ t\ a =$
$cost\ t + w_b \times depth\ t\ a$
else
if $b \in alphabet\ t$ *then*
$cost\ (swapLeaves\ t\ w_a\ a\ w_b\ b) + freq\ t\ b \times depth\ t\ b =$

```
      cost t + w_a × depth t b
  else
    cost (swapLeaves t w_a a w_b b) = cost t
proof (induct t)
  case Leaf show case by simp
next
  case (InnerNode w t_1 t_2)
  note c = ‹consistent (InnerNode w t_1 t_2)›
  note hyps = InnerNode
  have w_1: if a ∈ alphabet t_1 then
           if b ∈ alphabet t_1 then
             weight (swapLeaves t_1 w_a a w_b b) + freq t_1 a + freq t_1 b =
               weight t_1 + w_a + w_b
             else
             weight (swapLeaves t_1 w_a a w_b b) + freq t_1 a = weight t_1 + w_b
         else
           if b ∈ alphabet t_1 then
             weight (swapLeaves t_1 w_a a w_b b) + freq t_1 b = weight t_1 + w_a
           else
             weight (swapLeaves t_1 w_a a w_b b) = weight t_1 using hyps
    by (simp add: weight_swapLeaves)
  have w_2: if a ∈ alphabet t_2 then
           if b ∈ alphabet t_2 then
             weight (swapLeaves t_2 w_a a w_b b) + freq t_2 a + freq t_2 b =
               weight t_2 + w_a + w_b
           else
             weight (swapLeaves t_2 w_a a w_b b) + freq t_2 a = weight t_2 + w_b
         else
           if b ∈ alphabet t_2 then
             weight (swapLeaves t_2 w_a a w_b b) + freq t_2 b = weight t_2 + w_a
           else
             weight (swapLeaves t_2 w_a a w_b b) = weight t_2 using hyps
    by (simp add: weight_swapLeaves)
  show case
  proof cases
    assume a_1: a ∈ alphabet t_1
    hence a_2: a ∉ alphabet t_2 using c by auto
    show case
    proof cases
      assume b_1: b ∈ alphabet t_1
      hence b ∉ alphabet t_2 using c by auto
      thus case using a_1 a_2 b_1 w_1 w_2 hyps by simp
    next
      assume b_1: b ∉ alphabet t_1 show case
```

proof *cases*
assume $b \in$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
next
assume $b \notin$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
qed
qed
next
assume a_1: $a \notin$ *alphabet* t_1 **show** *case*
proof *cases*
assume a_2: $a \in$ *alphabet* t_2 **show** *case*
proof *cases*
assume b_1: $b \in$ *alphabet* t_1
hence $b \notin$ *alphabet* t_2 **using** c **by** *auto*
thus *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
next
assume b_1: $b \notin$ *alphabet* t_1 **show** *case*
proof *cases*
assume $b \in$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
next
assume $b \notin$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
qed
qed
next
assume a_2: $a \notin$ *alphabet* t_2 **show** *case*
proof *cases*
assume b_1: $b \in$ *alphabet* t_1
hence $b \notin$ *alphabet* t_2 **using** c **by** *auto*
thus *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
next
assume b_1: $b \notin$ *alphabet* t_1 **show** *case*
proof *cases*
assume $b \in$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
next
assume $b \notin$ *alphabet* t_2 **thus** *case* **using** a_1 a_2 b_1 w_1 w_2 *hyps* **by** *simp*
qed
qed
qed
qed
qed

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 (induct t)
  case Leaf thus case by 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 t1)
      case (Leaf w_c c)
      note l1 = Leaf
      show thesis
      proof (cases height t2 = 0)
        case True
        note h2 = True
        show thesis
        proof (cases t2)
          case Leaf thus thesis using l1 hyps by auto metis+
        next
          case InnerNode thus thesis using h2 by simp
        qed
      next
        case False
        note h2 = False
        show thesis
        proof cases
          assume c = b thus thesis using l1 h2 hyps by simp
        next
          assume c ≠ b
          have sibling t2 b ∈ alphabet t2 using ‹c ≠ b› l1 h2 hyps
            by (simp add: sibling_ne_imp_sibling_in_alphabet)
          thus thesis using ‹c ≠ b› l1 h2 hyps by auto
        qed
      qed
    next
      case InnerNode thus thesis using h1 by simp
    qed
  next
    case False
```

note $h_1 = False$
show *thesis*
proof (*cases height* $t_2 = 0$)
  case *True*
  note $h_2 = True$
  show *thesis*
  proof (*cases* t_2)
    case (*Leaf* w_d *d*)
    note $l_2 = Leaf$
    show *thesis*
    proof *cases*
      assume $d = b$ **thus** *thesis* **using** h_1 l_2 *hyps* **by** *simp*
    next
      assume $d \neq b$ **show** *thesis*
      proof (*cases* $b \in alphabet\ t_1$)
        case *True*
        hence *sibling* t_1 $b \in alphabet\ t_1$ **using** ‹$d \neq b$› h_1 l_2 *hyps*
          by (*simp add*: *sibling_ne_imp_sibling_in_alphabet*)
        thus *thesis* **using** *True* ‹$d \neq b$› h_1 l_2 *hyps*
          by (*simp add*: *alphabet_swapLeaves*)
      next
        case *False* **thus** *thesis* **using** ‹$d \neq b$› l_2 *hyps* **by** *simp*
      qed
    qed
  next
    case *InnerNode* **thus** *thesis* **using** h_2 **by** *simp*
  qed
next
  case *False*
  note $h_2 = False$
  show *thesis*
  proof (*cases* $b \in alphabet\ t_1$)
    case *True* **thus** *thesis* **using** h_1 h_2 *hyps* **by** *auto*
  next
    case *False*
    note $b_1 = False$
    show *thesis*
    proof (*cases* $b \in alphabet\ t_2$)
      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

qed
qed
qed
qed

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*
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* =

$\quad$ *cost t* $+$ *freq t a* $\times$ *depth t b* $+$ *freq t b* $\times$ *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*:
$[\![i \le j;\ m \le (n::nat)]\!] \Longrightarrow i \times n + j \times m \le i \times m + j \times n$
apply $(subgoal\_tac\ i \times m + i \times (n - m) + j \times m \le i \times m + j \times m + j \times (n - m))$
apply (*simp add*: *diff_mult_distrib*2)
by *simp*

lemma *cost_swapSyms_le*:
assumes *consistent t* $a \in$ *alphabet t* $b \in$ *alphabet t freq t a* $\le$ *freq t b*
depth t a $\le$ *depth t b*
shows *cost* (*swapSyms t a b*) $\le$ *cost t*
proof $-$
let *aabb* $=$ *freq t a* $\times$ *depth t a* $+$ *freq t b* $\times$ *depth t b*
let *abba* $=$ *freq t a* $\times$ *depth t b* $+$ *freq t b* $\times$ *depth t a*
have *abba* $\le$ *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 nat_add_assoc* [*THEN sym*])
also have $\ldots \le$ *cost t* $+$ *aabb* **using** ‹*abba* $\le$ *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*]:
$[\![$*consistent t*; *sibling t b* $\ne b$; $a \ne b]\!] \Longrightarrow$
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*]:
$[\![$*consistent t*; *sibling t b* $\ne a$; *sibling t b* $\ne b$; $a \ne b]\!] \Longrightarrow$
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 *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 $a \neq b$ 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

$$\mathit{swapFourSyms}\ t\ a\ b\ c\ d \equiv \mathit{swapSyms}\ (\mathit{swapSyms}\ t\ a\ c)\ b\ d.$$

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. Incidentally, Cormen et al. [3, p. 390] forgot to consider this case in their proof.[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*
by (*simp add*: *swapFourSyms_def*)

lemma *consistent_swapFourSyms* [*simp*]:
consistent t ⟹ *consistent* (*swapFourSyms t a b c d*)
by (*simp add*: *swapFourSyms_def*)

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*
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."

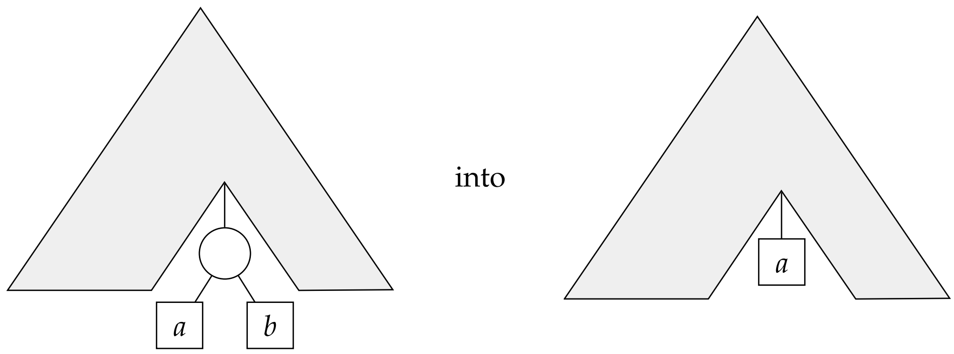
lemma *sibling_swapFourSyms_when_4th_is_sibling*:
assumes *consistent t a* ∈ *alphabet t b* ∈ *alphabet t c* ∈ *alphabet t*
    a ≠ *b sibling t c* ≠ *c*
shows *sibling* (*swapFourSyms t a b c* (*sibling t c*)) *a* = *b*
proof (*cases a* ≠ *sibling t c* ∧ *b* ≠ *c*)

[2] Thomas Cormen indicated in a personal communication that this will be corrected in the next edition of the book.

case *True* **show** *thesis*
proof −
let $d = sibling\ t\ c$
let $t_s = swapFourSyms\ t\ a\ b\ c\ d$
have *abba*: $(sibling\ t_s\ a = b) = (sibling\ t_s\ b = a)$ **using** ‹*consistent t*›
by (*metis consistent_swapFourSyms sibling_reciprocal*)
have *s*: $sibling\ t\ c = sibling\ (swapSyms\ t\ a\ c)\ a$ **using** *True assms*
by (*metis sibling_reciprocal sibling_swapSyms_sibling*)
have $sibling\ t_s\ b = sibling\ (swapSyms\ t\ a\ c)\ d$ **using** *s True assms*
by (*auto simp*: *swapFourSyms_def*)
also have $\ldots = a$ **using** *True assms*
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
case *False* **thus** *thesis* **using** *assms*
by (*auto intro*: *sibling_reciprocal simp*: *swapFourSyms_def*)
qed

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* :: $'a\ tree \Rightarrow 'a \Rightarrow 'a\ tree$ **where**
$mergeSibling\ (Leaf\ w_b\ b)\ a = Leaf\ w_b\ b$
$mergeSibling\ (InnerNode\ w\ (Leaf\ w_b\ b)\ (Leaf\ w_c\ c))\ a =$
$(if\ a = b \lor a = c\ then\ Leaf\ (w_b + w_c)\ a$
$else\ InnerNode\ w\ (Leaf\ w_b\ b)\ (Leaf\ w_c\ c))$
$mergeSibling\ (InnerNode\ w\ t_1\ t_2)\ a =$

$InnerNode\ w\ (mergeSibling\ t_1\ a)\ (mergeSibling\ t_2\ a)$

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*]:
$a \notin alphabet\ t \Longrightarrow mergeSibling\ t\ a = t$
by (*induct t a rule*: *mergeSibling.induct*) *simp+*

lemma *height_gt_0_imp_mergeSibling_left* [*simp*]:
$height\ t_1 > 0 \Longrightarrow$
$mergeSibling\ (InnerNode\ w\ t_1\ t_2)\ a =$
$\quad InnerNode\ w\ (mergeSibling\ t_1\ a)\ (mergeSibling\ t_2\ a)$
by (*case_tac* t_1) *simp+*

lemma *height_gt_0_imp_mergeSibling_right* [*simp*]:
$height\ t_2 > 0 \Longrightarrow$
$mergeSibling\ (InnerNode\ w\ t_1\ t_2)\ a =$
$\quad InnerNode\ w\ (mergeSibling\ t_1\ a)\ (mergeSibling\ t_2\ a)$
by (*case_tac* t_2) *simp+*

lemma *either_height_gt_0_imp_mergeSibling* [*simp*]:
$height\ t_1 > 0 \lor height\ t_2 > 0 \Longrightarrow$
$mergeSibling\ (InnerNode\ w\ t_1\ t_2)\ a =$
$\quad InnerNode\ w\ (mergeSibling\ t_1\ a)\ (mergeSibling\ t_2\ a)$
by *auto*

lemma *alphabet_mergeSibling* [*simp*]:
$[\![consistent\ t;\ a \in alphabet\ t]\!] \Longrightarrow$
$alphabet\ (mergeSibling\ t\ a) = (alphabet\ t - \{sibling\ t\ a\}) \cup \{a\}$
by (*induct t a rule*: *mergeSibling_induct_consistent*) *auto*

lemma *consistent_mergeSibling* [*simp*]:
$consistent\ t \Longrightarrow consistent\ (mergeSibling\ t\ a)$
by (*induct t a rule*: *mergeSibling_induct_consistent*) *auto*

lemma *freq_mergeSibling*:
$[\![consistent\ t;\ a \in alphabet\ t;\ sibling\ t\ a \neq a]\!] \Longrightarrow$
$freq\ (mergeSibling\ t\ a) =$
$\quad (\lambda c.\ if\ c = a\ then\ freq\ t\ a + freq\ t\ (sibling\ t\ a)$
$\qquad else\ if\ c = sibling\ t\ a\ then\ 0$

else freq t c)
apply (*rule ext*)
apply (*induct t a rule*: *mergeSibling_induct_consistent*)
by *auto*

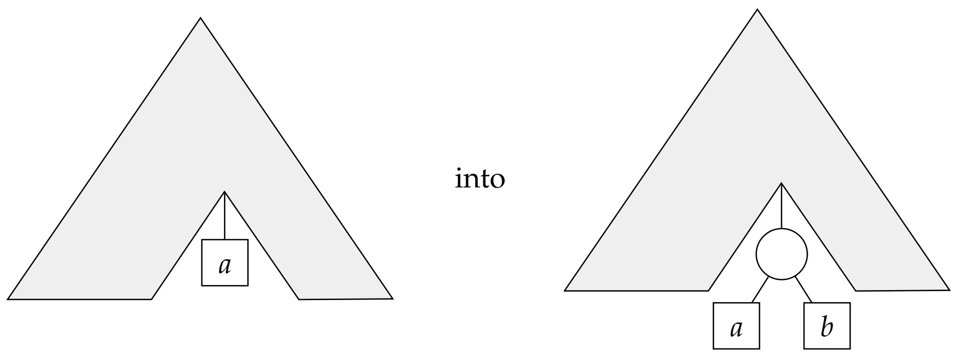
lemma *weight_mergeSibling* [*simp*]:
$weight\ (mergeSibling\ t\ a) = weight\ t$
by (*induct t a rule*: *mergeSibling.induct*) *simp+*

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*:
$[\![consistent\ t;\ sibling\ t\ a \neq a]\!] \Longrightarrow$
$cost\ (mergeSibling\ t\ a) + freq\ t\ a + freq\ t\ (sibling\ t\ a) = cost\ t$
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



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**
$splitLeaf\ (Leaf\ w_c\ c)\ w_a\ a\ w_b\ b =$
  $(if\ c = a\ then\ InnerNode\ w_c\ (Leaf\ w_a\ a)\ (Leaf\ w_b\ b)\ else\ Leaf\ w_c\ c)$
$splitLeaf\ (InnerNode\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b =$
  $InnerNode\ w\ (splitLeaf\ t_1\ w_a\ a\ w_b\ b)\ (splitLeaf\ t_2\ w_a\ a\ w_b\ b)$

primrec $splitLeaf_F$ :: *'a forest* ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a* ⇒ *'a forest* **where**
$splitLeaf_F\ []\ w_a\ a\ w_b\ b = []$
$splitLeaf_F\ (t \cdot ts)\ w_a\ a\ w_b\ b =$
  $splitLeaf\ t\ w_a\ a\ w_b\ b \cdot splitLeaf_F\ ts\ w_a\ a\ w_b\ b$

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

lemma *notin_alphabet_imp_splitLeaf_id* [*simp*]:
$a \notin \mathit{alphabet}\ t \Longrightarrow \mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b = t$
by (*induct t*) *simp+*

lemma *notin_alphabet*$_F$*_imp_splitLeaf*$_F$*_id* [*simp*]:
$a \notin \mathit{alphabet}_F\ \mathit{ts} \Longrightarrow \mathit{splitLeaf}_F\ \mathit{ts}\ w_a\ a\ w_b\ b = \mathit{ts}$
by (*induct ts*) *simp+*

lemma *alphabet_splitLeaf* [*simp*]:
$\mathit{alphabet}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) =$
$\quad (\mathit{if}\ a \in \mathit{alphabet}\ t\ \mathit{then}\ \mathit{alphabet}\ t \cup \{b\}\ \mathit{else}\ \mathit{alphabet}\ t)$
by (*induct t*) *simp+*

lemma *consistent_splitLeaf* [*simp*]:
$[\![\mathit{consistent}\ t;\ b \notin \mathit{alphabet}\ t]\!] \Longrightarrow \mathit{consistent}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b)$
by (*induct t*) *auto*

lemma *freq_splitLeaf* [*simp*]:
$[\![\mathit{consistent}\ t;\ b \notin \mathit{alphabet}\ t]\!] \Longrightarrow$
$\mathit{freq}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) =$
$\quad (\mathit{if}\ a \in \mathit{alphabet}\ t\ \mathit{then}$
$\quad\quad (\lambda c.\ \mathit{if}\ c = a\ \mathit{then}\ w_a\ \mathit{else}\ \mathit{if}\ c = b\ \mathit{then}\ w_b\ \mathit{else}\ \mathit{freq}\ t\ c)$
$\quad\ \mathit{else}$
$\quad\quad \mathit{freq}\ t)$
apply (*rule ext*)
apply (*induct t b rule*: *tree_induct_consistent*)
by *auto*

lemma *weight_splitLeaf* [*simp*]:
$[\![\mathit{consistent}\ t;\ a \in \mathit{alphabet}\ t;\ \mathit{freq}\ t\ a = w_a + w_b]\!] \Longrightarrow$
$\mathit{weight}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) = \mathit{weight}\ t$
by (*induct t a rule*: *tree_induct_consistent*) *simp+*

lemma *cost_splitLeaf* [*simp*]:
$[\![\mathit{consistent}\ t;\ a \in \mathit{alphabet}\ t;\ \mathit{freq}\ t\ a = w_a + w_b]\!] \Longrightarrow$
$\mathit{cost}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) = \mathit{cost}\ t + w_a + w_b$
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**
$sortedByWeight\ [] = True$
$sortedByWeight\ [t] = True$
$sortedByWeight\ (t_1 \cdot t_2 \cdot ts) =$
$\quad (weight\ t_1 \leq weight\ t_2 \wedge sortedByWeight\ (t_2 \cdot ts))$

The function obeys the following fairly obvious laws.

lemma *sortedByWeight_Cons_imp_sortedByWeight*:
$sortedByWeight\ (t \cdot ts) \Longrightarrow sortedByWeight\ ts$
by (*case_tac ts*) *simp+*

lemma *sortedByWeight_Cons_imp_forall_weight_ge*:
$sortedByWeight\ (t \cdot ts) \Longrightarrow \forall u \in set\ ts.\ weight\ u \geq weight\ t$
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*:
$[\![sortedByWeight\ ts;\ height\ t = 0;\ height_F\ ts = 0]\!] \Longrightarrow$
$sortedByWeight\ (insortTree\ t\ ts)$
by (*induct ts rule*: *sortedByWeight.induct*) *auto*

4.8 Pair of Minimal Symbols

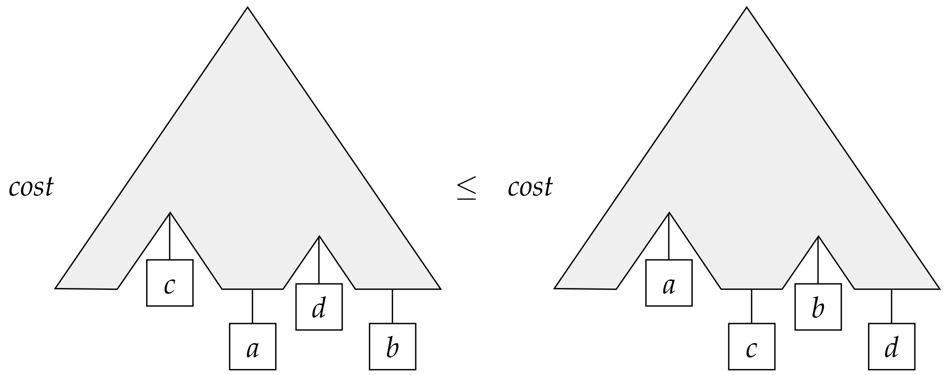
The *minima* predicate expresses that two symbols $a, b \in alphabet\ t$ have the lowest frequencies in the tree t and that $freq\ t\ a \leq freq\ t\ b$. Minimal symbols need not be uniquely defined.

definition *minima* :: *'a tree* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
$minima\ t\ a\ b \equiv$
$\quad a \in alphabet\ t \wedge b \in alphabet\ t \wedge a \neq b \wedge freq\ t\ a \leq freq\ t\ b$
$\quad \wedge (\forall c \in alphabet\ t.\ c \neq a \longrightarrow c \neq b \longrightarrow$
$\qquad\qquad freq\ t\ c \geq freq\ t\ a \wedge freq\ t\ c \geq freq\ t\ b)$

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



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. [3, p. 389] expresses a similar property, which turns out to be a corollary of our cost property:

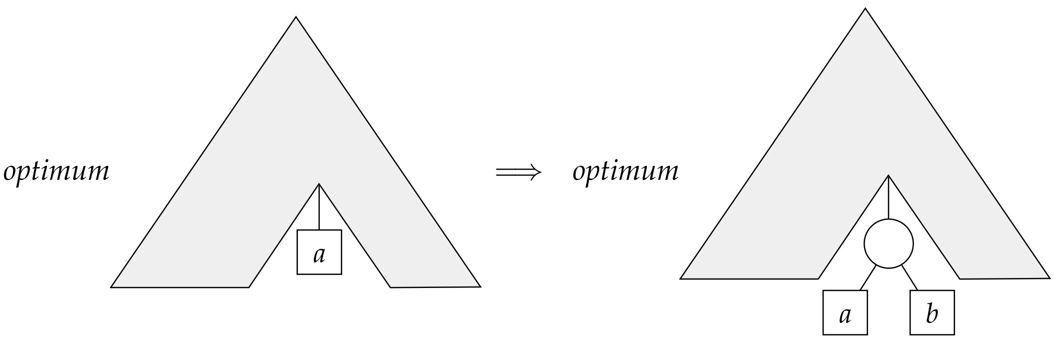
> Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

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 *lems* $=$ *swapFourSyms_def minima_def cost_swapSyms_le depth_le_height*
 show *thesis*
 proof (*cases* $a \neq d \wedge b \neq c$)
  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: lems)
    next
      assume b ≠ d thus thesis using ‹a = c› True assms
        by (simp add: lems)
    qed
  next
    assume a ≠ c show thesis
    proof cases
      assume b = d thus thesis using ‹a ≠ c› True assms
        by (simp add: lems)
    next
      assume b ≠ d
      have cost (swapFourSyms t a b c d) ≤ cost (swapSyms t a c)
        using ‹b ≠ d› ‹a ≠ c› True assms by (clarsimp simp: lems)
      also have ... ≤ cost t using ‹b ≠ d› ‹a ≠ c› True assms
        by (clarsimp simp: lems)
      finally show thesis .
    qed
  qed
next
  case False thus thesis using assms by (clarsimp simp: lems)
qed
qed
```

5.2 Leaf Split Optimality Lemma

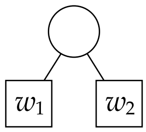
The tree *splitLeaf t* w_a *a* w_b *b* is optimum if *t* is optimum, under a few assumptions, notably that *a* and *b* are minima of the new tree and that *freq t a* $= w_a + w_b$. Graphically:



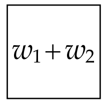
This corresponds to the following fragment of Knuth's proof:

> Now it is easy to prove that the weighted path length of such a tree is

minimized if and only if the tree with



replaced by

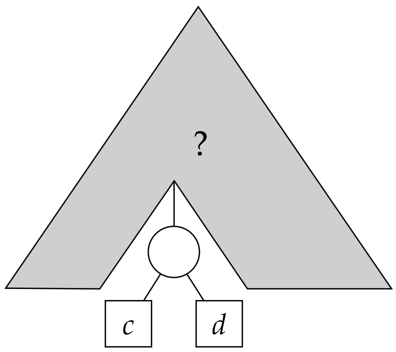


has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$.

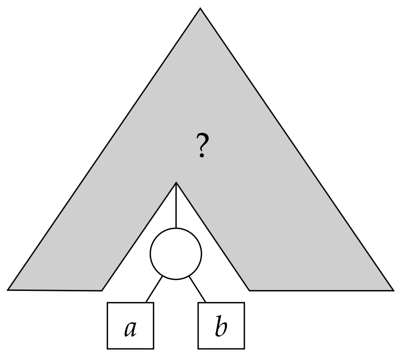
(We only need the "if" direction of Knuth's equivalence.) Lemma 16.3 in Cormen et al. [3, p. 391] expresses essentially the same property:

> 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.

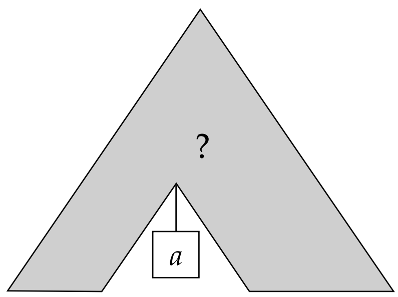
The proof is as follows: We assume that t has a cost less than or equal to that of any other compatible tree v and show that *splitLeaf* t w_a a w_b b has a cost less than or equal to that of any other compatible tree u. By *exists_at_height* and *depth_height_imp_sibling_ne*, we know that some symbols 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 compatible with *t*, which we can use to instantiate *v* in the assumption:



With this instantiation, the proof is easy:

$$
\begin{aligned}
& \mathit{cost}\ (\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b) \\
= \quad & \hfill (\mathit{cost\_splitLeaf}) \\
& \mathit{cost}\ t + w_a + w_b \\
\le \quad & \hfill (\text{assumption}) \\
& \mathit{cost}\ (\overbrace{\mathit{mergeSibling}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d)\ a}^{v}) + w_a + w_b \\
= \quad & \hfill (\mathit{cost\_mergeSibling}) \\
& \mathit{cost}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d) \\
\le \quad & \hfill (\mathit{cost\_swapFourSyms\_le}) \\
& \mathit{cost}\ u.
\end{aligned}
$$

The proof in Cormen et al. is by contradiction: Essentially, they assume that there exists a tree *u* with a lower cost than *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*.

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

lemma *twice_freq_le_imp_minima*:
$[\![\forall c \in \mathit{alphabet}\ t.\ w_a \leq \mathit{freq}\ t\ c \wedge w_b \leq \mathit{freq}\ t\ c;$
$\mathit{alphabet}\ u = \mathit{alphabet}\ t \cup \{b\};\ a \in \mathit{alphabet}\ u;\ a \neq b;$
$\mathit{freq}\ u = (\lambda c.\ \mathit{if}\ c = a\ \mathit{then}\ w_a\ \mathit{else}\ \mathit{if}\ c = b\ \mathit{then}\ w_b\ \mathit{else}\ \mathit{freq}\ t\ c);$
$w_a \leq w_b]\!] \Longrightarrow$
minima u a b
by (*simp add*: *minima_def*)

Now comes the key lemma.

lemma *optimum_splitLeaf*:
assumes *consistent t optimum t* $a \in \mathit{alphabet}\ t$ $b \notin \mathit{alphabet}\ t$
$\mathit{freq}\ t\ a = w_a + w_b$ $\forall c \in \mathit{alphabet}\ t.\ \mathit{freq}\ t\ c \geq w_a \wedge \mathit{freq}\ t\ c \geq w_b$
$w_a \leq w_b$
shows *optimum* $(\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b)$
proof (*unfold optimum_def*, *clarify*)
fix *u*
let $t' = \mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b$
assume c_u: *consistent u*
and a_u: $\mathit{alphabet}\ t' = \mathit{alphabet}\ u$
and f_u: $\mathit{freq}\ t' = \mathit{freq}\ u$
show $\mathit{cost}\ t' \leq \mathit{cost}\ u$
proof $(\mathit{cases}\ \mathit{height}\ t' = 0)$
case *True* **thus** *thesis* **by** *simp*
next
case *False*
hence h_u: $\mathit{height}\ u > 0$ **using** a_u *assms*
by (*auto intro*: *height_gt_0_alphabet_eq_imp_height_gt_0*)
have a_a: $a \in \mathit{alphabet}\ u$ **using** a_u *assms* **by** *fastsimp*
have a_b: $b \in \mathit{alphabet}\ u$ **using** a_u *assms* **by** *fastsimp*
have *ab*: $a \neq b$ **using** *assms* **by** *blast*
from *exists_at_height* [*OF* c_u]
obtain *c* **where** a_c: $c \in \mathit{alphabet}\ u$ **and** d_c: $\mathit{depth}\ u\ c = \mathit{height}\ u$ **..**
let $d = \mathit{sibling}\ u\ c$
have *dc*: $d \neq c$ **using** d_c c_u h_u a_c **by** (*rule depth_height_imp_sibling_ne*)
have a_d: $d \in \mathit{alphabet}\ u$ **using** *dc*
by (*rule sibling_ne_imp_sibling_in_alphabet*)
have d_d: $\mathit{depth}\ u\ d = \mathit{height}\ u$ **using** d_c c_u **by** *simp*

let $u' = \mathit{swapFourSyms}\ u\ a\ b\ c\ d$
have $c_{u'}$: *consistent* u' **using** c_u **by** *simp*
have $a_{u'}$: $\mathit{alphabet}\ u' = \mathit{alphabet}\ u$ **using** a_a a_b a_c a_d a_u **by** *simp*
have $f_{u'}$: $\mathit{freq}\ u' = \mathit{freq}\ u$ **using** a_a a_b a_c a_d c_u f_u **by** *simp*
have s_a: $\mathit{sibling}\ u'\ a = b$ **using** c_u a_a a_b a_c *ab dc*
by (*rule sibling_swapFourSyms_when_4th_is_sibling*)

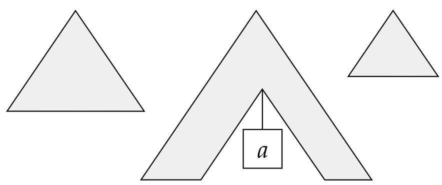
let $v = \mathit{mergeSibling}\ u'\ a$
have c_v: *consistent v* **using** $c_{u'}$ **by** *simp*
have a_v: $\mathit{alphabet}\ v = \mathit{alphabet}\ t$ **using** $s_a\ c_{u'}\ a_{u'}\ a_a\ a_u$ *assms* **by** *auto*
have f_v: $\mathit{freq}\ v = \mathit{freq}\ t$
  using $s_a\ c_{u'}\ a_{u'}\ f_{u'}\ f_u$ [*THEN sym*] *ab* a_u [*THEN sym*] *assms*
  by (*simp add*: *freq_mergeSibling ext*)

have $\mathit{cost}\ t' = \mathit{cost}\ t + w_a + w_b$ **using** *assms* **by** *simp*
also have $\ldots \leq \mathit{cost}\ v + w_a + w_b$ **using** $c_v\ a_v\ f_v$ ‹*optimum t*›
  by (*simp add*: *optimum_def*)
also have $\ldots = \mathit{cost}\ u'$
  proof $-$
    have $\mathit{cost}\ v + \mathit{freq}\ u'\ a + \mathit{freq}\ u'\ (\mathit{sibling}\ u'\ a) = \mathit{cost}\ u'$
      using $c_{u'}\ s_a$ *assms* **by** (*subst cost_mergeSibling*) *auto*
    moreover have $w_a = \mathit{freq}\ u'\ a\ \ w_b = \mathit{freq}\ u'\ b$
      using $f_{u'}\ f_u$ [*THEN sym*] *assms* **by** *clarsimp+*
    ultimately show *thesis* **using** s_a **by** *simp*
  qed
also have $\ldots \leq \mathit{cost}\ u$
  proof $-$
    have *minima u a b* **using** $a_u\ f_u$ *assms*
      by (*subst twice_freq_le_imp_minima*) *auto*
    with c_u **show** *thesis* **using** $a_c\ a_d\ d_c\ d_d$ *dc* [*THEN not_sym*]
      by (*rule cost_swapFourSyms_le*)
  qed
finally show *thesis* **.**
qed
qed

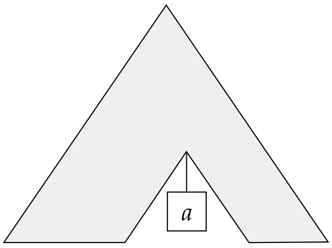
5.3 Leaf Split Commutativity Lemma

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:

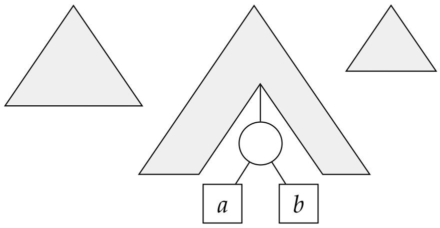
(1)



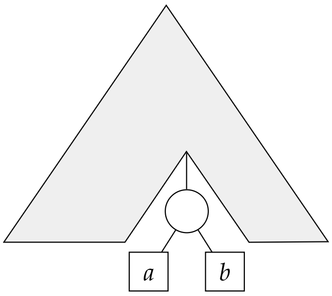
(2a)



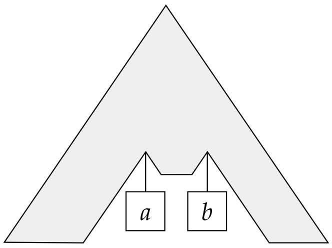
(2b)



(3a)



(3b)



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

$$\mathit{splitLeaf}\ (\mathit{huffman}\ \mathit{ts})\ w_a\ a\ w_b\ b = \mathit{huffman}\ (\mathit{splitLeaf}_F\ \mathit{ts}\ w_a\ a\ w_b\ b)$$

when *ts* is consistent, $a \in \mathit{alphabet}_F\ \mathit{ts}$, $b \notin \mathit{alphabet}_F\ \mathit{ts}$, and $\mathit{freq}_F\ \mathit{ts}\ a = w_a + w_b$. But before we can prove this commutativity lemma, we need to introduce a few simple lemmas.

lemma *cachedWeight_splitLeaf* [*simp*]:
$\mathit{cachedWeight}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) = \mathit{cachedWeight}\ t$
by (*induct t*) *simp*+

lemma *splitLeaf*$_F$*_insortTree_when_in_alphabet_left* [*simp*]:
$[\![a \in \mathit{alphabet}\ t;\ \mathit{consistent}\ t;\ a \notin \mathit{alphabet}_F\ \mathit{ts};\ \mathit{freq}\ t\ a = w_a + w_b]\!] \Longrightarrow$
$\mathit{splitLeaf}_F\ (\mathit{insortTree}\ t\ \mathit{ts})\ w_a\ a\ w_b\ b = \mathit{insortTree}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b)\ \mathit{ts}$
by (*induct ts*) *simp*+

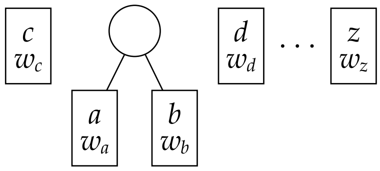
lemma *splitLeaf*$_F$*_insortTree_when_in_alphabet*$_F$*_tail* [*simp*]:
$[\![a \in \mathit{alphabet}_F\ \mathit{ts};\ \mathit{consistent}_F\ \mathit{ts};\ a \notin \mathit{alphabet}\ t;\ \mathit{freq}_F\ \mathit{ts}\ a = w_a + w_b]\!] \Longrightarrow$
$\mathit{splitLeaf}_F\ (\mathit{insortTree}\ t\ \mathit{ts})\ w_a\ a\ w_b\ b =$
  $\mathit{insortTree}\ t\ (\mathit{splitLeaf}_F\ \mathit{ts}\ w_a\ a\ w_b\ b)$
proof (*induct ts*)
 case *Nil* **thus** *case* **by** *simp*
next
 case (*Cons u us*) **show** *case*
 proof (*cases* $a \in \mathit{alphabet}\ u$)
  case *True*

moreover hence $a \notin alphabet_F\ us$ **using** *Cons* **by** *auto*
ultimately show *thesis* **using** *Cons* **by** *auto*
next
case *False* **thus** *thesis* **using** *Cons* **by** *simp*
qed
qed

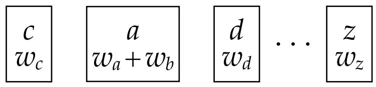
We are now ready to prove the commutativity lemma.

lemma *splitLeaf_huffman_commute*:
$[\![consistent_F\ ts;\ ts \neq [\,];\ a \in alphabet_F\ ts;\ freq_F\ ts\ a = w_a + w_b]\!] \Longrightarrow$
$splitLeaf\ (huffman\ ts)\ w_a\ a\ w_b\ b = huffman\ (splitLeaf_F\ ts\ w_a\ a\ w_b\ b)$
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* $a \in alphabet\ t_1$)
case *True*
moreover hence $a \notin alphabet\ t_2$ $a \notin alphabet_F\ ts$ **using** *hyps* **by** *auto*
ultimately show *thesis* **using** *hyps* **by** (*simp add*: *uniteTrees_def*)
next
case *False*
note $a_1 = False$
show *thesis*
proof (*cases* $a \in alphabet\ t_2$)
case *True*
moreover hence $a \notin alphabet_F\ ts$ **using** *hyps* **by** *auto*
ultimately show *thesis* **using** a_1 *hyps* **by** (*simp add*: *uniteTrees_def*)
next
case *False*
thus *thesis* **using** a_1 *hyps* **by** *simp*
qed
qed
qed

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. This effectively provides a way to flatten the forest at each step of the algorithm. Its invocation is implicit in the textbook proof.

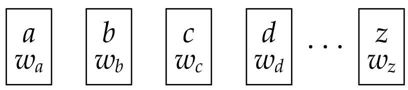
5.4 Optimality Theorem

We are one lemma away from our main result.

lemma *max_0_imp_0* [*simp*]:
$(max\ x\ y = (0{::}nat)) = (x = 0 \wedge y = 0)$
by *auto*

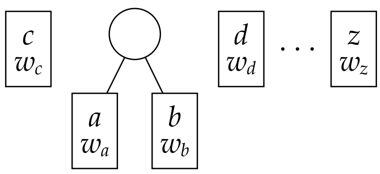
theorem *optimum_huffman*:
$[\![consistent_F\ ts;\ height_F\ ts = 0;\ sortedByWeight\ ts;\ ts \neq [\,]]\!] \Longrightarrow$
optimum (*huffman ts*)

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

huffman



(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

$$splitLeaf\left(huffman\ \boxed{\begin{matrix}c\\w_c\end{matrix}}\ \boxed{\begin{matrix}a\\w_a+w_b\end{matrix}}\ \boxed{\begin{matrix}d\\w_d\end{matrix}}\cdots\boxed{\begin{matrix}z\\w_z\end{matrix}}\right)\ w_a\ a\ w_b\ b.$$

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

$$huffman\ \boxed{\begin{matrix}c\\w_c\end{matrix}}\ \boxed{\begin{matrix}a\\w_a+w_b\end{matrix}}\ \boxed{\begin{matrix}d\\w_d\end{matrix}}\cdots\boxed{\begin{matrix}z\\w_z\end{matrix}}$$

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* ≠ []⟩ **by** *fast*
next
case (*Cons* t_a *ts*′)
note *ts* = *Cons*
show *thesis*
proof (*cases ts*′)
case *Nil* **thus** *thesis* **using** *ts hyps* **by** (*simp add*: *optimum_def*)
next
case (*Cons* t_b *ts*″)
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* = *insortTree* (*uniteTrees* t_a t_b) *ts*″
let *us*′ = *insortTree* (*Leaf* (w_a + w_b) *a*) *ts*″
let t_s = *splitLeaf* (*huffman us*′) w_a *a* w_b *b*
have e_1: *huffman ts* = *huffman us* **using** *ts*′ *ts* **by** *simp*
have e_2: *huffman us* = t_s **using** l_a l_b *ts*′ *ts hyps*
by (*auto simp*: *splitLeaf_huffman_commute uniteTrees_def*)

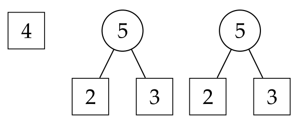
```
          have optimum (huffman us') using l_a ts' ts hyps
            by (drule_tac x = us' in spec)
              (auto dest: sortedByWeight_Cons_imp_sortedByWeight
                 simp: sortedByWeight_insortTree)
          hence optimum t_s using l_a l_b ts' ts hyps
            apply simp
            apply (rule optimum_splitLeaf)
            by (auto dest!: height_F_0_imp_Leaf_freq_F_in_set
                       sortedByWeight_Cons_imp_forall_weight_ge)
          thus optimum (huffman ts) using e_1 e_2 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
qed

end
```

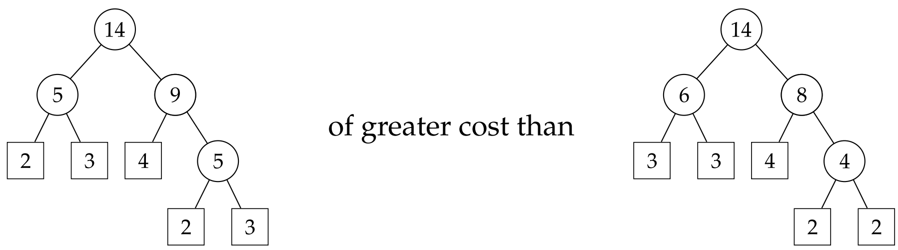
Theorem *optimum_huffman* assumes that the forest *ts* passed to *huffman* consists exclusively of leaf nodes. It is tempting to relax this restriction, by requiring instead that the forest *ts* is optimum. We would define optimality of a forest as follows:

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

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



6 Related Work

Laurent Théry's Coq formalization of Huffman's algorithm [11, 12] is an obvious yardstick for our work. It has a somewhat wider scope, proving among others the isomorphism between prefix codes and 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 subtrees.

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 then 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 without too much work:

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.

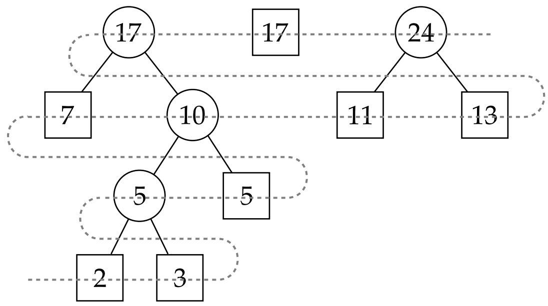
4. By restricting our attention to consistent trees, we were able to define the *depth* function simply and meaningfully.

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 was to demonstrate that the textbook proof of Huffman's algorithm can be formalized in a straightforward manner 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. [3]. Concerning Isabelle, the main lesson to draw from the Huffman proof is 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 + well-formedness predicate" combination (*'a tree* + *consistent*) and functions defined by sequential pattern matching (*sibling* and *mergeSibling*).

In addition, formalizing the proof of Huffman's algorithm 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:



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 next step could be to extend the proof's scope to to cover *encode/decode* functions and connect prefix code trees 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 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] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms* (Second Edition). MIT Press, 2001 and McGraw-Hill, 2002.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] 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.