J Autom Reasoning (2009) 43:1–18
DOI 10.1007/s10817-009-9116-y

Proof Pearl: Mechanizing the Textbook Proof of Huffman's Algorithm

Jasmin Christian Blanchette

Received: 10 October 2008 / Accepted: 27 January 2009 / Published online: 13 February 2009

Abstract Huffman's algorithm is a procedure for constructing a binary tree with minimum weighted path length. Our Isabelle/HOL 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.

Keywords Huffman coding · Interactive theorem provers · Higher-order logic

1 Introduction

Huffman's algorithm [9] is a simple and elegant procedure for constructing a binary tree with minimum weighted path length—a measure of cost that considers both the lengths of the paths from the root to the leaf nodes and weights associated with the leaf nodes. The algorithm's main application is data compression: By equating leaf nodes with characters and weights with character frequencies, we can use it to derive optimum character codes.

This paper presents a formalization of the correctness proof of Huffman's algorithm written using Isabelle/HOL 2008 [13]. Our proof is based on the informal proofs given by Knuth [10] and Cormen et al. [6]. The development was done independently of Laurent Théry's Coq proof [14, 15], which through its "cover" concept represents a considerable departure from the textbook proof. The complete development comprises 95 propositions and is available online as part of the Archive of Formal Proofs [4].

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

J. C. Blanchette (✉)
Institut für Informatik, Technische Universität München, Munich, Germany
e-mail: blanchette@in.tum.de

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to Isabelle/HOL. Section 3 introduces Huffman's algorithm, and Section 4 presents a functional implementation. Section 5 reviews the informal textbook proof. Sections 6 and 7 develop a small library of functions needed for the formal proof. Section 8 presents the key lemmas and theorems of the formal proof. Section 9 describes the custom induction rules developed specifically for this proof. Section 10 compares our work with Théry's Coq proof. Finally, Section 11 concludes the paper.

2 Isabelle/HOL

Isabelle is a generic theorem prover whose built-in metalogic is an intuitionistic fragment of higher-order logic [7, 13]. Isabelle's HOL object logic provides a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers.

The term language consists of simply typed λ-terms written in an ML-like syntax [12]. 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 $x + y$. Variables may range over functions and predicates.

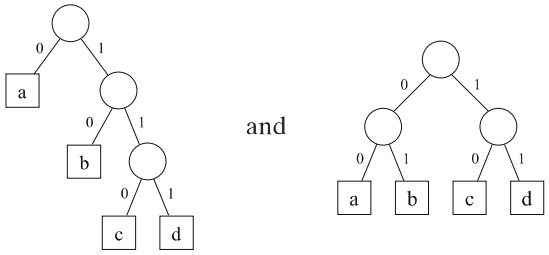
The type of lists over α consists of the empty list [] and the infix constructor $x \cdot xs$, where x has type α and xs is a list over α. Sets are written using traditional notation.

3 Huffman's Algorithm

Suppose we want to encode strings over a finite source alphabet as sequences of bits. Fixed-length codes like ASCII are simple and fast, but they generally waste space. If we know the frequency w_a of each source symbol a, we can save space by using shorter code words for the most frequent symbols. We say that a (variable-length) code is *optimum* if it minimizes the sum $\sum_a w_a \delta_a$, where δ_a is the length of the binary code word for a.

As an example, consider the source string 'abacabad'. Encoding 'abacabad' with the code $C_1 = \{\text{a} \mapsto 0, \text{b} \mapsto 10, \text{c} \mapsto 110, \text{d} \mapsto 111\}$ gives the 14-bit code word 01001100100111. The code C_1 is optimum: No code that unambiguously encodes source symbols one at a time could do better than C_1 on the input 'abacabad'. With a fixed-length code such as $C_2 = \{\text{a} \mapsto 00, \text{b} \mapsto 01, \text{c} \mapsto 10, \text{d} \mapsto 11\}$ we need at least 16 bits to encode the same string.

Binary codes can be represented by binary trees. For example, the trees



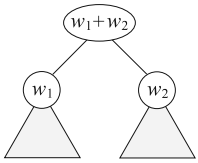
correspond to C_1 and C_2. The code word for a given symbol can be obtained as follows: Start at the root and descend toward the leaf node associated with the symbol one node at a time; generate a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The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

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no code word is a prefix of another. It is sufficient to consider only full binary trees (trees whose inner nodes all have two children), because any node with only one child can advantageously be eliminated by removing it and letting the child take its parent's place.

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. In diagrams, the nodes are often annotated with their weights.

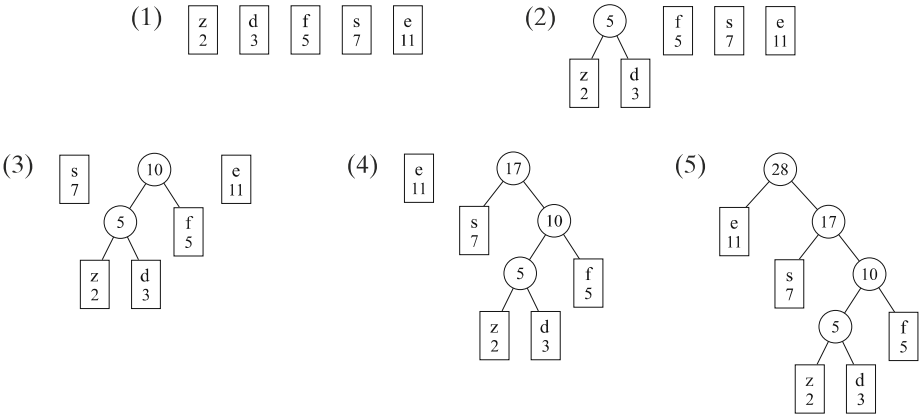
David Huffman [9] discovered a simple algorithm for constructing an optimum code tree for specified symbol frequencies: Create a forest consisting of only leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as initial weights for the nodes. Then pick the two trees

with the lowest weights and replace them with the tree



Repeat this process until only one tree is left.

As an illustration, executing the algorithm for the frequencies $f_\mathrm{d} = 3$, $f_\mathrm{e} = 11$, $f_\mathrm{f} = 5$, $f_\mathrm{s} = 7$, $f_\mathrm{z} = 2$ gives rise to the following sequence of states:



Tree (5) is optimum for the given frequencies.

4 Functional Implementation of the Algorithm

The functional implementation of the algorithm [3, pp. 242–244] relies on an α *tree* datatype:

$$\textbf{datatype}\ \alpha\ tree\ =\ \mathit{Leaf}\ nat\ \alpha\ |\ \mathit{InnerNode}\ nat\ (\alpha\ tree)\ (\alpha\ tree).$$

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. The *cachedWeight* function extracts the weight stored in a node:

$$\mathit{cachedWeight}\ (\mathit{Leaf}\ w\ a) = w; \qquad \mathit{cachedWeight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = w.$$

The implementation builds on two additional auxiliary functions. The first one, *uniteTrees*, combines two trees by adding an inner node above them:

$$\mathit{uniteTrees}\ t_1\ t_2\ =\ \mathit{InnerNode}\ (\mathit{cachedWeight}\ t_1 + \mathit{cachedWeight}\ t_2)\ t_1\ t_2.$$

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

$$\begin{aligned}\mathit{insortTree}\ u\ [\,]\ &=\ [u]\\ \mathit{insortTree}\ u\ (t \cdot ts)\ &=\ (\text{if}\ \mathit{cachedWeight}\ u \le \mathit{cachedWeight}\ t\ \text{then}\ u \cdot t \cdot ts\\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{else}\ t \cdot \mathit{insortTree}\ u\ ts).\end{aligned}$$

The main function that implements Huffman's algorithm follows:

$$\begin{aligned}\mathit{huffman}\ [t]\ &=\ t\\ \mathit{huffman}\ (t_1 \cdot t_2 \cdot ts)\ &=\ \mathit{huffman}\ (\mathit{insortTree}\ (\mathit{uniteTrees}\ t_1\ t_2)\ ts).\end{aligned}$$

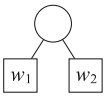
The function should initially be invoked with a list of leaf nodes sorted by weight. It repeatedly unites the first two trees of the forest it receives as argument until a single tree is left. The time complexity of the algorithm is quadratic in the size of the forest.

5 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 [10, pp. 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 $w_1 \le w_2 \le w_3 \le \cdots \le w_m$, where $m \ge 2$, and suppose that we are given a tree that minimizes the weighted path length. (Such a tree certainly exists, since only finitely many binary trees with *m* terminal nodes are possible.) Let *V* be an internal node of maximum distance from the root. If w_1 and w_2 are not the weights already attached to the children of *V*, we can interchange them with the values that are already there; such an interchange does not increase the weighted path length.

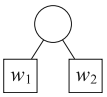
Thus there is a tree that minimizes the weighted path length and contains the subtree



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

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

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



Indeed, the formalization works without it. Cormen et al. [6, pp. 385–391] provide a 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.*

Lemma 16.4 *Procedure* Huffman *produces an optimal prefix code.*

6 Basic Auxiliary Functions

The formal proof is distributed over four sections. This section formally defines basic concepts such as alphabet, consistency, and optimality, which are needed to specify the correctness of Huffman's algorithm. The next section introduces more specialized functions that arise in the proof. Section 8 presents the central lemmas and theorems, which echo the textbook proof. Finally, Section 9 explains the use of custom induction rules.

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

$$alphabet\ (Leaf\ w\ a) = \{a\};\quad alphabet\ (InnerNode\ w\ t_1\ t_2) = alphabet\ t_1 \cup alphabet\ t_2.$$

For sets and predicates, Isabelle supports both inductive definitions and recursive functions. We consistently favor recursion over induction, partly because recursion gives rise to simplification rules that greatly help automatic proof tactics, and partly because Isabelle's counterexample generator **quickcheck** [2], which we used extensively during the top-down development of the proof, has better support for recursive definitions.

A tree is *consistent* if for each inner node the alphabets of the two subtrees are disjoint. Intuitively, this means that a symbol occurs in at most one leaf node. Consistency is a sufficient condition for δ_a (the length of the *unique* code word for a) to be defined. Although this wellformedness property is not mentioned in algorithms textbooks [1, 6, 10], it is essential and appears as an assumption in many of our lemmas. The definition follows:

$$\begin{aligned} consistent\ (Leaf\ w\ a) &= True \\ consistent\ (InnerNode\ w\ t_1\ t_2) &= (consistent\ t_1 \wedge consistent\ t_2 \\ &\quad \wedge alphabet\ t_1 \cap alphabet\ t_2 = \emptyset). \end{aligned}$$

The *depth* of a symbol (which we denoted by δ_a in Section 3) is the length of the path from the root to that symbol, or equivalently the length of the code word for the symbol:

$$\begin{aligned} depth\ (Leaf\ w\ b)\ a &= 0 \\ depth\ (InnerNode\ w\ t_1\ t_2)\ a &= (\text{if } a \in alphabet\ t_1 \text{ then } depth\ t_1\ a + 1 \\ &\quad \text{else if } a \in alphabet\ t_2 \text{ then } depth\ t_2\ a + 1 \\ &\quad \text{else } 0). \end{aligned}$$

Symbols that do not occur in the tree or that occur at the root of a one-node tree have depth 0. If a symbol occurs in several leaf nodes, the depth is arbitrarily defined in terms of the leftmost node labeled with that symbol. The definition may seem very inefficient from a functional programming point of view, but this does not matter, because unlike Huffman's algorithm, the *depth* function is merely a reasoning tool and is never actually executed.

The *height* of a tree is the length of the longest path from the root to a leaf node, or equivalently the length of the longest code word:

$$\begin{aligned} height\ (Leaf\ w\ a) &= 0 \\ height\ (InnerNode\ w\ t_1\ t_2) &= max\ (height\ t_1)\ (height\ t_2) + 1. \end{aligned}$$

The *frequency* of a symbol (which we denoted by w_a in Section 3) is the sum of the weights attached to the leaf nodes labeled with that symbol:

$$\begin{aligned} freq\ (Leaf\ w\ b)\ a &= (\text{if } a = b \text{ then } w \text{ else } 0) \\ freq\ (InnerNode\ w\ t_1\ t_2)\ a &= freq\ t_1\ a + freq\ t_2\ a. \end{aligned}$$

For consistent trees, the sum comprises at most one nonzero term. The frequency is then the weight of the leaf node labeled with the symbol, or 0 if there is no such node.

Two trees are *comparable* if they have the same alphabet and symbol frequencies. This is an important concept, because it allows us to state not only that the tree constructed by Huffman's algorithm is optimal but also that it has the expected alphabet and frequencies.

The *weight* function returns the weight of a tree:

$$\begin{aligned} \mathit{weight}\ (\mathit{Leaf}\ w\ a) &= w \\ \mathit{weight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) &= \mathit{weight}\ t_1 + \mathit{weight}\ t_2. \end{aligned}$$

In the *InnerNode* case, we ignore the weight cached in the node and instead compute the tree's weight recursively. This makes reasoning simpler because we can then avoid specifying cache correctness as an assumption in our lemmas. Equivalently, we can define the weight as the sum of the symbol frequencies: $\mathit{weight}\ t = \sum_{a\,\in\,\mathit{alphabet}\ t} \mathit{freq}\ t\ a$.

The *cost* (or *weighted path length*) of a consistent tree is the sum $\sum_{a\,\in\,\mathit{alphabet}\ t} \mathit{freq}\ t\ a \times \mathit{depth}\ t\ a$ (which we denoted by $\sum_a w_a \delta_a$ in Section 3). It obeys the recursive law

$$\begin{aligned} \mathit{cost}\ (\mathit{Leaf}\ w\ a) &= 0 \\ \mathit{cost}\ (\mathit{InnerNode}\ w\ t_1\ t_2) &= \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2. \end{aligned}$$

A tree is optimum if and only if its cost is not greater than that of any comparable tree:

$$\begin{aligned} \mathit{optimum}\ t\ =\ (&\forall u.\ \mathit{consistent}\ u \longrightarrow \mathit{alphabet}\ t = \mathit{alphabet}\ u \longrightarrow \mathit{freq}\ t = \mathit{freq}\ u \\ &\longrightarrow \mathit{cost}\ t \leq \mathit{cost}\ u). \end{aligned}$$

Tree functions are readily generalized to forests; for example, the alphabet of a forest is defined as the union of the alphabets of its trees. The forest generalizations have an '$_F$' attached to their name (for example, $\mathit{alphabet}_F$). The definitions are given in the Appendix.

7 Other Functions Needed for the Formal Proof

The textbook proof interchanges nodes in trees, replaces a two-leaf tree with weights w_1 and w_2 by a single leaf of weight $w_1 + w_2$ and vice versa, and refers to the two symbols with the lowest frequencies. In the formalization, these concepts are represented by four functions: *swapFourSyms*, *mergeSibling*, *splitLeaf*, and *minima*.

The four-way symbol interchange function *swapFourSyms* 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 *swapSyms* (*swapSyms t a c*) *b d*, where *swapSyms* (defined in the Appendix) exchanges two symbols. This naive

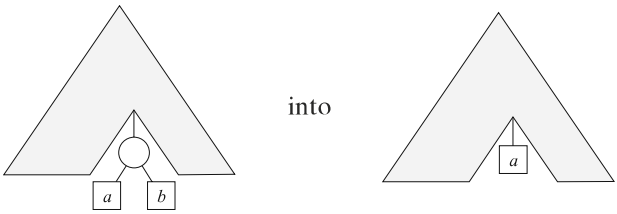
definition fails in the face of aliasing: If $a = d$, but $b \neq c$, then *swapFourSyms a b c d* would leave a in b's position.[1] A better definition is

$$\begin{aligned} \mathit{swapFourSyms}\ t\ a\ b\ c\ d\ =\ & (\text{if } a = d \text{ then } \mathit{swapSyms}\ t\ b\ c \\ & \text{else if } b = c \text{ then } \mathit{swapSyms}\ t\ a\ d \\ & \text{else } \mathit{swapSyms}\ (\mathit{swapSyms}\ t\ a\ c)\ b\ d). \end{aligned}$$

The following lemma about *swapSyms* captures the intuition that more frequent symbols should be encoded using fewer bits than less frequent ones to minimize the cost:

Lemma 7.1 (Symbol Interchange Cost) *If consistent t, $a \in$ alphabet t, $b \in$ alphabet t, freq t a $\leq$ freq t b, and depth t a $\leq$ depth t b, then cost* (*swapSyms t a b*) $\leq$ *cost t.*

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



The frequency of a in the resulting tree is the sum of the original frequencies of a and b. The function is defined by the equations

$$\begin{aligned} \mathit{mergeSibling}\ (\mathit{Leaf}\ w_b\ b)\ a &= \mathit{Leaf}\ w_b\ b \\ \mathit{mergeSibling}\ (\mathit{InnerNode}\ w\ \ (\mathit{Leaf}\ w_b\ b)\ \ (\mathit{Leaf}\ w_c\ c))\ a &= \begin{array}{l}(\text{if } a = b \vee a = c \text{ then } \mathit{Leaf}\ (w_b + w_c)\ a \\ \ \text{else } \mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c))\end{array} \\ \mathit{mergeSibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a &= \mathit{InnerNode}\ w\ \ (\mathit{mergeSibling}\ t_1\ a) \\ & \qquad\qquad\qquad\qquad (\mathit{mergeSibling}\ t_2\ a). \end{aligned}$$

As in ML, the defining equations are applied sequentially [11]; that is, the third equation is applicable only if the second does not match.

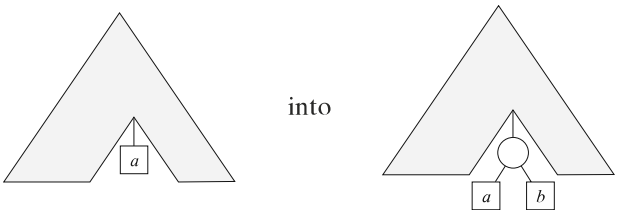
The *sibling* function returns the label of the node that is the (left or right) sibling of the node labeled with the given symbol a in tree t. If a is not in t's alphabet or it occurs in a node with no sibling leaf, we simply return a. This gives us the nice property that if t is consistent, then *sibling t a* $\neq a$ if and only if a has a sibling. The definition, given in the Appendix, distinguishes the same cases as *mergeSibling*.

Using the *sibling* function, we can state that merging two sibling leaves with weights w_a and w_b decreases the cost by $w_a + w_b$:

Lemma 7.2 (Sibling Merge Cost) *If consistent t and sibling t a* $\neq a$, *then cost* (*mergeSibling t a*) + *freq t a* + *freq t* (*sibling t a*) = *cost t.*

[1] Cormen et al. [6, p. 390] did not account for this scenario in their proof. Thomas Cormen indicated in a personal communication that this will be corrected in the next edition of the book.

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 *splitLeaf* with w_a and w_b such that *freq t a* $= w_a + w_b$. The definition follows:

$$\begin{aligned} \mathit{splitLeaf}\ (\mathit{Leaf}\ w_c\ c)\ w_a\ a\ w_b\ b\ &=\ (\text{if } c = a\ \text{ then } \mathit{InnerNode}\ w_c\ \ (\mathit{Leaf}\ w_a\ a) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Leaf}\ w_b\ b) \\ &\qquad\qquad\qquad\qquad\quad \text{else}\ \ \mathit{Leaf}\ w_c\ c) \\ \mathit{splitLeaf}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b\ &=\ \mathit{InnerNode}\ w\ \ (\mathit{splitLeaf}\ t_1\ w_a\ a\ w_b\ b) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{splitLeaf}\ t_2\ w_a\ a\ w_b\ b). \end{aligned}$$

Splitting a leaf with weight $w_a + w_b$ into two sibling leaves with weights w_a and w_b increases the cost by $w_a + w_b$:

Lemma 7.3 (Leaf Split Cost) *If consistent t, a* $\in$ *alphabet t, and freq t a* $= w_a + w_b$, *then cost* (*splitLeaf t* w_a *a* w_b *b*) = *cost t* $+ w_a + w_b$.

Finally, the *minima* predicate expresses that two symbols a, b have the lowest frequencies in the tree t and that *freq t a* $\leq$ *freq t b*:

$$\begin{aligned} \mathit{minima}\ t\ a\ b\ \ =\ \ & a \in \mathit{alphabet}\ t \wedge b \in \mathit{alphabet}\ t \wedge a \neq b \wedge \mathit{freq}\ t\ a \leq \mathit{freq}\ t\ b \\ & \wedge\ (\forall c \in \mathit{alphabet}\ t.\ \ c \neq a \longrightarrow c \neq b \longrightarrow \\ & \qquad\qquad \mathit{freq}\ t\ a \leq \mathit{freq}\ t\ c \wedge \mathit{freq}\ t\ b \leq \mathit{freq}\ t\ c). \end{aligned}$$

8 The Key Lemmas and Theorems

It is easy to prove that the tree returned by Huffman's algorithm preserves the alphabet, consistency, and symbol frequencies of the original forest.

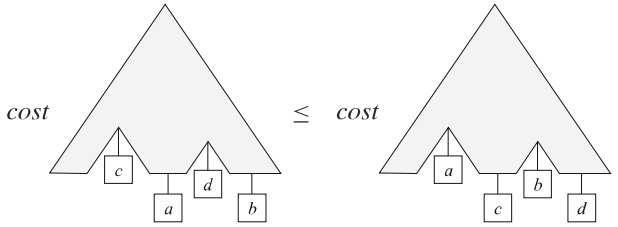
Theorem 8.1 (Huffman Alphabet) *If ts* $\neq$ [], *then alphabet* (*huffman ts*) = $\mathit{alphabet}_{\mathsf{F}}$ *ts*.

Theorem 8.2 (Huffman Consistency) *If* $\mathit{consistent}_{\mathsf{F}}$ *ts and ts* $\neq$ [], *then consistent* (*huffman ts*).

Theorem 8.3 (Huffman Frequencies) *If ts* $\neq$ [], *then freq* (*huffman ts*) = $\mathit{freq}_{\mathsf{F}}$ *ts*.

The main challenge is to prove the optimality of the tree constructed by Huffman's algorithm. We need three lemmas before we can present the optimality theorem. The first two lemmas correspond to Lemmas 16.2 and 16.3 in Cormen et al.

First, if a and b are minima, and c and d are at the very bottom of the tree, then exchanging a and b with c and d does not increase the tree's cost. Graphically, we have



This cost property is part of Knuth's proof:

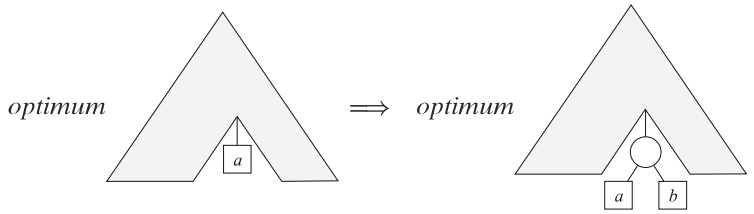
> Let V be an internal node of maximum distance from the root. If w_1 and w_2 are not the weights already attached to the children of V, we can interchange them with the values that are already there; such an interchange does not increase the weighted path length.

Lemma 16.2 in Cormen et al. expresses a similar property.

Lemma 8.4 (Four-Way Symbol Interchange Cost) *If consistent t, minima t a b, c ∈ alphabet t, d ∈ alphabet t, depth t c = height t, depth t d = height t, and c ≠ d, then cost* (*swapFourSyms t a b c d*) ≤ *cost t.*

Proof The proof is by case distinctions on $a = c$, $a = d$, $b = c$, and $b = d$. The cases are easy to prove by expanding *swapFourSyms* and applying Lemma 7.1. □

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



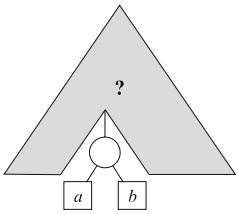
This corresponds to the following fragment of Knuth's proof:

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

We only need the "if" direction of Knuth's equivalence. Lemma 16.3 in Cormen et al. expresses essentially the same property.

Lemma 8.5 (Leaf Split Optimality) *If consistent t, optimum t,* $a \in$ *alphabet t,* $b \notin$ *alphabet t, freq t a* $= w_a + w_b$, $\forall c \in$ *alphabet t.* $w_a \leq$ *freq t c* $\wedge\ w_b \leq$ *freq t c, and* $w_a \leq w_b$, *then optimum* (*splitLeaf t* w_a *a* w_b *b*).

Proof We assume that *t*'s cost is less than or equal to that of any other comparable tree *v* and show that *splitLeaf t* w_a *a* w_b *b* has a cost less than or equal to that of any other comparable tree *u*. For the nontrivial case where *height t* > 0, it is easy to prove that there must be two symbols *c* and *d* occurring in sibling nodes at the very bottom of *u*. From *u* we construct the tree *swapFourSyms u a b c d* in which the minima *a* and *b* are siblings:



The question mark is there to remind us that we know nothing specific about *u*'s structure. Merging *a* and *b* gives a tree comparable with *t*, which we can use to instantiate *v*:[2]

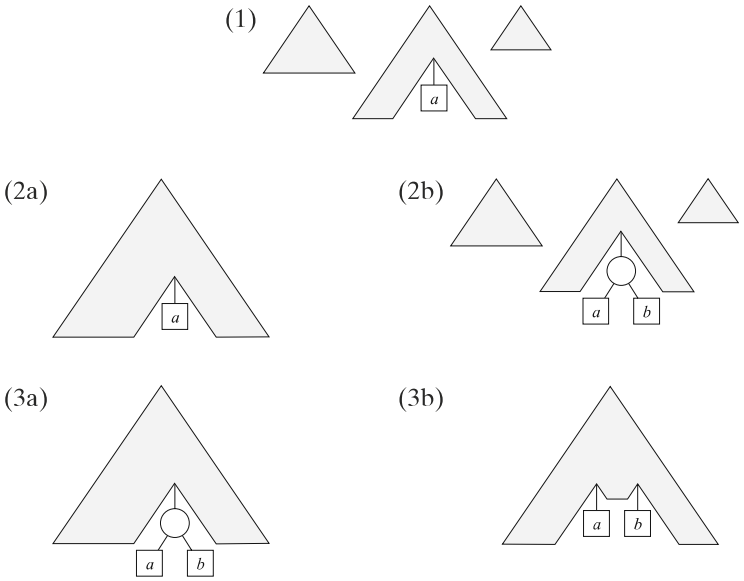
$$
\begin{aligned}
& \mathit{cost}\ (\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b) \\
= \quad & && \text{by Lemma 7.3} \\
& \mathit{cost}\ t + w_a + w_b \\
\leq \quad & && \text{by assumption} \\
& \mathit{cost}\ (\overbrace{\mathit{mergeSibling}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d)\ a}^{v})\triangleleft + w_a + w_b \\
= \quad & && \text{by Lemma 7.2} \\
& \mathit{cost}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d) \\
\leq \quad & && \text{by Lemma 8.4} \\
& \mathit{cost}\ u.
\end{aligned}
$$

□

A key property of Huffman's algorithm is that once it has combined two lowest-weight trees using *uniteTrees*, it does not visit these trees ever again. This suggests that splitting a leaf node before applying the algorithm should give the same result as

[2] In contrast, the proof in Cormen et al. is by contradiction: Essentially, they assume that there exists a tree *u* with a lower cost than *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 Lemma 8.4, they invoke their Lemma 16.2, which is questionable since *u* is not necessarily optimum. Thomas Cormen commented that this step will be clarified in the next edition of the book.

applying the algorithm first and splitting the leaf node afterward. The diagram below illustrates the situation:

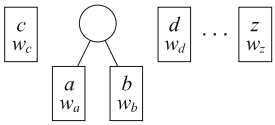


From the original forest (1), we can either run the algorithm (2a) and then split a (3a) or split a (2b) and then run the algorithm (3b). Lemma 8.6 asserts that the trees (3a) and (3b) are identical.

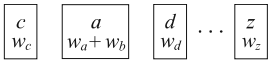
Lemma 8.6 (Leaf Split Commutativity) *If* $consistent_{\mathsf{F}}\ ts$, $ts \neq [\,]$, *and* $a \in alphabet_{\mathsf{F}}\ ts$, *then* $splitLeaf\ (huffman\ ts)\ w_a\ a\ w_b\ b = huffman\ (splitLeaf_{\mathsf{F}}\ ts\ w_a\ a\ w_b\ b)$.

Proof The proof is by straightforward induction on the length of the forest ts. □

An important consequence of this 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

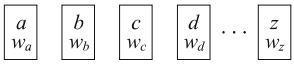


followed by splitting the leaf node a into two nodes a and b with frequencies w_a, w_b. The lemma effectively provides a way to flatten the forest at each step of the algorithm. Its invocation is implicit in the textbook proof.

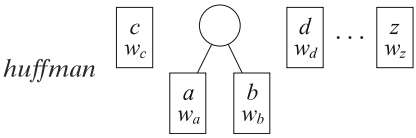
This leads us to our main result.

Theorem 8.7 (Huffman Optimality) *If consistent*$_{\mathsf{F}}$ *ts, height*$_{\mathsf{F}}$ *ts* $= 0$*, sortedByWeight ts, and ts* $\neq [\,]$*, then optimum* (*huffman ts*).

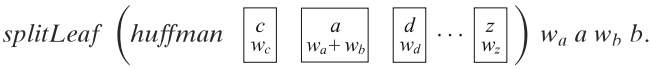
Proof The proof is by induction on the length of *ts*. The assumptions ensure that *ts* is of the form



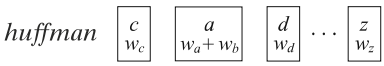
with $w_a \leq w_b \leq w_c \leq w_d \leq \cdots \leq w_z$. (The definition of *sortedByWeight* is given in the Appendix.) If *ts* consists of a single node, the node has cost 0 and is therefore optimum. If *ts* has length 2 or more, the first step of the algorithm leaves us with the term



In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By Lemma 8.6, the above tree equals



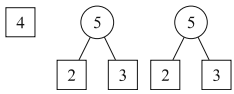
To prove that this tree is optimum, it suffices by Lemma 8.5 to show that



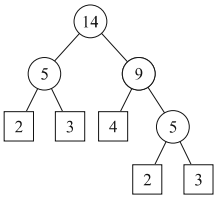
is optimum, which follows from the induction hypothesis. □

So what have we achieved? Assuming that our definitions really mean what we intend them to mean, we established that our functional implementation of Huffman's algorithm, when invoked properly, constructs a binary tree that represents an optimal prefix code for the specified alphabet and frequencies. Using Isabelle's code generator [8], we can convert the Isabelle code into Standard ML, OCaml, or Haskell and use it in a real application.

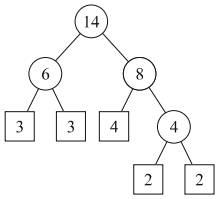
As a side note, Theorem 8.7 assumes that the forest *ts* passed to *huffman* consists exclusively of leaf nodes. It is tempting to relax this restriction, by requiring instead that the forest *ts* has the lowest cost among forests of the same size. However, the modified proposition does not hold. A counterexample is the optimum forest



for which the algorithm constructs the tree



of greater cost than



9 Custom Induction Rules

Higher-order logic makes it possible to define custom induction rules, which save us from writing repetitive proof scripts and help Isabelle's automatic proof tactics. We use them to distinguish cases that the standard structural induction rules treat as one case, to collapse cases that the standard rules needlessly distinguish, and to introduce additional assumptions that help the simplifier.

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

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

Instead of performing the above case distinction manually each time, we encode it in an induction rule:

Lemma 9.1 (Structural Induction Rule for Consistent Trees)
$[\![\mathit{consistent}\ t;$
$\bigwedge w_b\ b\ a.\ P\ (\mathit{Leaf}\ w_b\ b)\ a;$
$\bigwedge w\ t_1\ t_2\ a.\ [\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset;$
$\quad a \in \mathit{alphabet}\ t_1;\ a \notin \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$\quad P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.\ [\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset;$
$\quad a \notin \mathit{alphabet}\ t_1;\ a \in \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$\quad P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a;$
$\bigwedge w\ t_1\ t_2\ a.\ [\![\mathit{consistent}\ t_1;\ \mathit{consistent}\ t_2;\ \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset;$
$\quad a \notin \mathit{alphabet}\ t_1;\ a \notin \mathit{alphabet}\ t_2;\ P\ t_1\ a;\ P\ t_2\ a]\!] \Longrightarrow$
$\quad P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a]\!] \Longrightarrow$
$P\ t\ a.$

Lemma 9.1 involves two of Isabelle's metalogical operators: universal quantification, written $\bigwedge x_1\ \ldots\ x_n.\ \psi$ ("for all $x_1, \ldots, x_n$ we have ψ"), and implication, written $[\![\varphi_1; \ldots; \varphi_n]\!] \Longrightarrow \psi$ ("if φ_1 and $\ldots$ and φ_n, then ψ"). The lemma can be proved by

performing a standard structural induction on t and proceeding by cases—a straightforward but long-winded process. A nicer approach relies on the recently introduced *induct_scheme* tactic, which reduces the putative induction rule to simpler proof obligations. Internally, it reuses the machinery that constructs the default induction rules. The resulting proof obligations concern (a) case completeness, (b) invariant preservation (in our case, tree consistency), and (c) wellfoundedness. For Lemma 9.1, the obligations (a) and (b) can be discharged using Isabelle's simplifier and classical reasoner, whereas (c) requires a single invocation of *lexicographic_order*, a tactic that was originally designed to prove termination of recursive functions [5, 11].

The other custom rule is for *mergeSibling* and *sibling*, which are defined using sequential pattern matching. The default rules are almost identical and distinguish 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 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 in Lemma 9.1 for consistent trees, with the same benefits for automation. These observations lead us to develop a custom induction rule that works for both *mergeSibling* and *sibling* and that distinguishes the following cases:

BASE CASE: $t = \mathit{Leaf}\ w\ b$.
INDUCTION STEP 1: $t = \mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c)$ 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 formal statement of the rule, which we omit, is similar to Lemma 9.1, except that we now have two induction steps instead of one. Using the custom induction rule, Lemma 7.2 and several others could be proved entirely by Isabelle's *auto* tactic.

10 Related Work

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

Théry identified the following difficulties in formalizing the textbook proof:

1. The leaf interchange process that brings the two minimal symbols together is tedious to formalize.
2. The sibling merging process requires introducing a new symbol for the merged node, which complicates the formalization.

3. The algorithm constructs the tree in a bottom-up fashion. While top-down procedures can usually be proved by structural induction, bottom-up procedures often require more sophisticated induction principles and larger invariants.
4. The informal proof relies on the notion of depth of a node. Defining this notion formally is problematic, because the depth can only be seen as a function if the tree is composed of distinct nodes.

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

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

1. Formalizing the leaf interchange did not prove overly tedious. Among our 95 propositions, 24 concern *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.

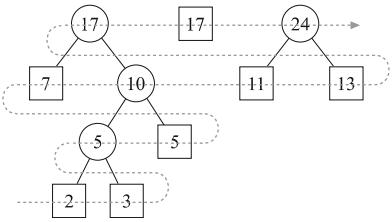
11 Conclusion

The main contribution of this paper has been to demonstrate that the textbook proof of Huffman's algorithm can be elegantly formalized using a state-of-the-art theorem prover such as Isabelle/HOL. In the process, we uncovered a few minor snags in the proof given in Cormen et al. [6].

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

Formalizing the proof of Huffman's algorithm also led to a deeper understanding of this classic algorithm. Many of the lemmas, notably Lemma 8.6, have not been found in the literature and express fundamental properties of the algorithm. The proof uncovered additional insights. In particular, each step of the algorithm appears

to preserve the invariant that the nodes in a forest are ordered by weight from left to right, bottom to top, as in the example below:



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

A few other directions for future work suggest themselves. First, we could formalize some of our hypotheses, notably our restriction to full and consistent binary trees. The current formalization says nothing about the algorithm's application for data compression, so a natural next step would be to extend the proof's scope to cover *encode*/*decode* functions and show that full binary trees are isomorphic to prefix codes.

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

Appendix: Additional Auxiliary Functions

$$
\begin{aligned}
alphabet_{\mathsf{F}}\ [\,] &= \emptyset & alphabet_{\mathsf{F}}\ (t \cdot ts) &= alphabet\ t \cup alphabet_{\mathsf{F}}\ ts\\
consistent_{\mathsf{F}}\ [\,] &= True & consistent_{\mathsf{F}}\ (t \cdot ts) &= (consistent\ t \wedge consistent_{\mathsf{F}}\ ts\\
& & & \quad \wedge\ alphabet\ t \cap alphabet_{\mathsf{F}}\ ts = \emptyset)\\
height_{\mathsf{F}}\ [\,] &= 0 & height_{\mathsf{F}}\ (t \cdot ts) &= max\ (height\ t)\ (height_{\mathsf{F}}\ ts)\\
freq_{\mathsf{F}}\ [\,]\ a &= 0 & freq_{\mathsf{F}}\ (t \cdot ts)\ a &= freq\ t\ a + freq_{\mathsf{F}}\ ts\ a\\
splitLeaf_{\mathsf{F}}\ [\,]\ w_a\ a\ w_b\ b &= [\,] & splitLeaf_{\mathsf{F}}\ (t \cdot ts)\ w_a\ a\ w_b\ b &= splitLeaf\ t\ w_a\ a\ w_b\ b\\
& & & \quad \cdot splitLeaf_{\mathsf{F}}\ ts\ w_a\ a\ w_b\ b
\end{aligned}
$$

$$
\begin{aligned}
swapLeaves\ (Leaf\ w_c\ c)\ w_a\ a\ w_b\ b &= (\text{if } c = a \text{ then } Leaf\ w_b\ b\\
&\quad \text{else if } c = b \text{ then } Leaf\ w_a\ a\\
&\quad \text{else } Leaf\ w_c\ c)\\
swapLeaves\ (InnerNode\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b &= InnerNode\ w\ (swapLeaves\ t_1\ w_a\ a\ w_b\ b)\\
&\quad (swapLeaves\ t_2\ w_a\ a\ w_b\ b)
\end{aligned}
$$

$$
swapSyms\ t\ a\ b = swapLeaves\ t\ (freq\ t\ a)\ a\ (freq\ t\ b)\ b
$$

$$
sibling\ (Leaf\ w_b\ b)\ a = a
$$

$$
\begin{aligned}
\mathit{sibling}\ (\mathit{InnerNode}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c))\ a &= (\text{if } a = b \text{ then } c \text{ else if } a = c \text{ then } b \text{ else } a)\\
\mathit{sibling}\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a &= (\text{if } a \in \mathit{alphabet}\ t_1 \text{ then } \mathit{sibling}\ t_1\ a\\
&\quad \text{else if } a \in \mathit{alphabet}\ t_2 \text{ then } \mathit{sibling}\ t_2\ a\\
&\quad \text{else } a)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{sortedByWeight}\ [\,] &= \mathit{True}\\
\mathit{sortedByWeight}\ [t] &= \mathit{True}\\
\mathit{sortedByWeight}\ (t_1 \cdot t_2 \cdot ts) &= \mathit{weight}\ t_1 \leq \mathit{weight}\ t_2 \wedge \mathit{sortedByWeight}\ (t_2 \cdot ts)
\end{aligned}
$$

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data Structures and Algorithms. Addison-Wesley, Reading (1983)
2. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) Software Engineering and Formal Methods (SEFM 2004), pp. 230–239. IEEE Computer Society, Los Alamitos (2004)
3. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead (1988)
4. Blanchette, J.C.: An Isabelle/HOL formalization of the textbook proof of Huffman's algorithm. Archive of Formal Proofs. http://afp.sourceforge.net/entries/Huffman.shtml (2008)
5. Bulwahn, L., Krauss, A.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT, Cambridge (2001)
7. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
8. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science, vol. 4732, pp. 128–143. Springer, Heidelberg (2007)
9. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proc. IRE 40(9), pp. 1098–1101 (1952)
10. Knuth, D.E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd edn. Addison-Wesley, Reading (1997)
11. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis (submitted), Dept. of Informatics, T. U. München (2009)
12. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised Edn.). MIT, Cambridge (1997)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, Heidelberg (2002) (Updated version http://isabelle.in.tum.de/doc/tutorial.pdf, 2008)
14. Théry, L.: A correctness proof of Huffman algorithm. http://coq.inria.fr/contribs/Huffman.html (2003). Accessed 24 September 2008
15. Théry, L.: Formalising Huffman's algorithm. Tech. report TRCS 034, Dept. of Informatics, Univ. of L'Aquila (2004)