Formalising Huffman's algorithm

Laurent Théry

Formalising Huffman's algorithm

L. Thery

Technical Report TRCS 034/2004

Formalising Huffman's algorithm

Laurent Théry
Dipartimento di Informatica
Università di L'Aquila, Italy

Abstract

This paper presents a formalisation of Huffman's algorithm that has been done in the Coq proof assistant. The result of the formalisation is a certified version of Huffman's algorithm written in the Ocaml functional programming language.

1 The algorithm

The algorithm is described in [2]. It generates an optimal code for encoding/decoding a text file. The premises are the following. The text file is composed of a sequence of characters. A code associates a sequence of bits to each character. Encoding consists in applying the association list to every character of the text to get the sequence of bits. Decoding is the opposite operation. Decoding only makes sense when the code is prefix free, i.e no sequence of bits in the association list is a prefix of any other sequence in the list. The code obtained using Huffman's algorithm is optimal in the sense that the size of the encoded file is as small as possible.

To explain the algorithm, we present its concise implementation in the functional language Ocaml [3]. As a text file is a sequence of characters, it can be represented by an object of type `char list`. The encoded file is a sequence of bits, so it can be represented by an object of type `bool list`. A code is an association list:

```
type code = (char * bool list) list
```

For example, the code that associates the sequence `101` to the character `a` and the sequence `0` to the character `b` is represented as:

```
[('a', [true; false; true]); ('b', [false])] : code
```

The coding function has the type:

```
encode:  code -> (char list) -> (bool list)
```

Encoding the text `bab` with the previous code returns the boolean list 01010. The decoding function has the following type:

```
decode:  code -> (bool list) -> (char list)
```

Using the previous code, decoding the list `00101` gives the text `bba`.

Huffman's algorithm uses the frequency of the different characters in the text to encode. The idea is simple: the more frequent a character is, the smaller its associated list of bits should be. In fact, to be closer to what the algorithm actually does we should say: the less frequent a character is, the longer its associated list of bits should be. The two sentences are of course equivalent.

To implement the algorithm we need two auxiliary functions. The first one computes the frequency of the different characters in the text:

```
frequency_list: (char list) -> ((char * int) list)
```

For example, given the text `aba` it returns the list

```
[('a', 2); ('b', 1)]
```

The second function inserts a pair, whose first element is an integer, into an ordered list by keeping the list ordered. Its type is

```
insert: (int * a) -> ((int * 'a) list) -> ((int * 'a) list)
```

A list is ordered if given any two consecutive pairs in the list, the first element of the first pair is smaller or equal to the first element of the second pair. For example, inserting `(3, "d")` in `[(0, "a"); (2, "b"); (4, "c")]` returns `[(0, "a"); (2, "b"); (3, "d"); (4, "c")]`,

Given these two functions, Huffman's algorithm can be implemented in OCAML in the following way:

```
let huffman m =
  let addl b l = map (fun (x, y) -> x, b :: y) l in
  let rec iter l = match l with
    | []      -> ([]: code)
    | [n, l] -> l
    | (n1, l1) :: (n2, l2) :: l ->
       iter
        (insert
            (n1 + n2, append (addl false l1) (addl true l2))
            l)
  in
    iter (fold_right (fun (a, n) l -> insert (n, [a, []]) l)
                     (frequency_list m)
                     [])
```

The self-contained implementation is given in Appendix A. To explain how the algorithm works, let us take the text `abbcccddddeeeee` as an example. Its frequency list is

```
[(a, 1); (b, 2); (c, 3); (d, 4); (e, 5)]
```

The algorithm recursively manipulates a list of type `(int * code) list`. Elements of this list are pairs that contain a code and its "frequency". During all the manipulations, the list is kept ordered from least "frequent" to most "frequent".

Initially the list is composed of singleton codes. For each character in the text, the singleton code associates an empty list of booleans to the character. The frequency of such singleton codes is equal to the frequency of the corresponding character in the text. In our example, we get the initial list:

```
[(1, [(a, [])]); (2, [(b, [])]); (3, [(c, [])]);
 (4, [(d, [])]); (5, [(e, [])])]
```

Now the algorithm iteratively takes the two least frequent codes and merges them together. This is done by appending the two lists, prefixing the lists of booleans of the first code by `false` and the lists of booleans of the second code by `true`. In our case, merging `[(a, [])]` and `[(b, [])` gives the new code

```
[(a, [false]); (b, [true])]
```

The frequency of the new code is the sum of the frequency of the two previous codes, so here 3. The new code and its frequency are then inserted so to keep the list ordered.

```
[(3, [(a, [false]); (b, [true])]); (3, [(c, [])]);
 (4, [(d, [])]); (5, [(e, [])])]
```

The process is iterated. At each step the length of the list is reduced by one. We first get a list with three elements by merging the code with `a`, `b` and the code with `c`:

```
[(4, [(d, [])]);  (5, [(e, [])]);
 (6, [(a, [false; false]); (b, [false; true]); (c, [true])])]
```

Then we merge the code with `d` and the code with `e`:

```
[(6, [(a, [false; false]); (b, [false; true]); (c, [true])]);
 (9, [(d, [false]); (e, [true])])]
```

Finally merging the code with `a`, `b`, `c` and the code with `d`, `e` leaves us with one single code and its frequency:

```
[(15, [(a, [false; false; false]);
       (b, [false; false; true]);
       (c, [false; true]);
       (d, [true; false]);
       (e, [true; true])])]
```

The resulting code is an optimal code.

2 The formalisation

In this section we present some aspects of the formalisation that has been done inside the Coq prover [1].

2.1 Codes

Codes are defined over an arbitrary alphabet `A`. A *code* is represented as an association list between elements of `A` and lists of booleans. A text is defined as a list of elements of `A`. In the following, we use the standard notations for lists: $[]$ represents the empty list, $[a;\ b]$ represents the list with two elements a and b. On lists we use the concatenation function $l_1 + l_2$, the function $length(l)$ that indicates the number of elements of a list, the predicate $a \in l$ that means that a belongs to l and the predicate $l_1 \approx l_2$ that means that l_1 is a permutation of l_2.

Two functions *encode* and *decode* are defined using association lists. These functions are total:

- when encoding, if a character of the text is not found in the code an empty list of booleans is generated;

- when decoding, if no prefix sequence of the encoded text corresponds to a sequence of booleans in the code the empty list of characters is generated.

Some predicates are defined on codes:

- A code c is *prefix free* iff

$$\forall (a_1,\ l_1) \in c, \forall (a_2,\ l_2) \in c,\ \ (l_1\ \mathit{prefix}\ l_2) \Rightarrow (a_1 = a_2 \land l_1 = l_2)$$

  where $(l_1\ \mathit{prefix}\ l_2)$ means that $\exists l_3,\ l_2 = l_1 + l_3$.

- A code c is *not null* iff

$$\forall (a,\ l) \in c,\ \ l \neq [].$$

- A code c is *in the alphabet* of a text m iff

$$\forall a \in m,\ \ \exists l,\ \ (a, l) \in c.$$

Some properties of the encoding and decoding operations are then derived. The main theorem proves that under some conditions the decoding operation is the reverse of the encoding one.

Theorem 2.1.1 (*correct Encoding*) *If c is not null, prefix free and in the alphabet of a text m, then $decode(c, encode(c, m)) = m$.*

Another theorem ensures that encoding does not depend on the order of the characters in the text.

Theorem 2.1.2 (*Encode Permutation*) *If $m_1 \approx m_2$, then encode(c, m_1) $\approx$ encode(c, m_2).*

The *weight* of a code c with respect to a text m is then defined as:

$$weight(m, c) \equiv length(encode(c, m)).$$

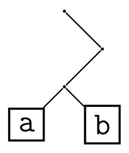
A code c is optimal for a text m iff c is not null, prefix free and in the alphabet of m and for all c_1 that is not null, prefix free and in the alphabet of m, $weight(c, m) \leq weight(c_1, m)$. What has been proved formally is that Huffman's algorithm always returns an optimal code.

2.2 Partial binary trees

In order to prove the algorithm correct we follow the usual proof on paper. It amounts to dealing with the so-called Huffman's trees. The first step is to consider partial binary trees. A partial binary tree is either a leaf ▭, a left node /, a right node \ or a binary node /\. It is relatively easy to show that there is an isomorphism between prefix free codes and partial binary trees with distinct leaves. For example, the code

```
[('a', [true; false; false]); ('b', [true; false; true])]
```

corresponds to the tree



To do the correspondence, we just need to interpret the list of booleans as a path in the tree structure: `true` as going down to the right and `false` as going down to the left.

Trees are a natural representation for codes. For example, the decoding operation can be easily explained using the tree structure. We start from the root of the tree. Reading one boolean tells us on which subtree to move down to. Once a leaf is reached, the character (the value of the leaf) is output and we start again from the root of the tree.

The isomorphism is formally proved. For this, we use two functions *buildTree* and *computeCode*. The former one takes a code and builds a tree. The latter one takes a tree and computes a code. We have the following two theorems that establish the isomorphism.

Theorem 2.2.1 (*Compute Build*) *If $c \neq []$ and is prefix free, then $computeCode(buildTree(c)) \approx c$.*

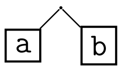
Theorem 2.2.2 (*Build Compute*) *If t is a partial binary tree with distinct leaves, then $buildTree(computeCode(t)) = t$.*

2.3 Binary trees

Binary trees are composed only of leaves $\square$ and binary nodes $\bigwedge$. A recursive function *toTree* transforms partial binary trees into binary trees by canceling left and right nodes:

$$
\begin{aligned}
toTree(\boxed{x}) &= \boxed{x} \\
toTree(\overset{/}{t}\ ) &= toTree(t) \\
toTree(\ \overset{\backslash}{\ t}) &= toTree(t) \\
toTree(\overset{/\backslash}{t_1\ t_2}) &= \overset{/\backslash}{toTree(t_1)\ \ toTree(t_2)}.
\end{aligned}
$$

When applied to the example of the previous section, it gives



As for partial binary trees, we can define a function *computeCode* that computes the corresponding code. With respect to the function *toTree* we have the following relation:

Theorem 2.3.1 (*toTree smaller*) *If t is a partial binary tree and m is a text, then $weight(m, computeCode(toTree(t))) \leq weight(m, computeCode(t))$.*

It follows that an optimal code can always be computed from a binary tree. So the initial problem of finding an optimal code can be expressed only using binary trees. For this, the notion of *weight* is also defined on binary trees. The definition takes as parameter a function f that gives the frequency of each character in the text m to encode. An auxiliary function *sum* that sums the frequencies of all the leaves of a tree is first defined:

$$
\begin{aligned}
sum(f, \boxed{x}) &= f(x) \\
sum(f, \overset{/\backslash}{t_1\ t_2}) &= sum(f, t_1)\ +\ sum(f, t_2).
\end{aligned}
$$

The weight function is then defined as:

$$\begin{array}{lcl} \mathit{weight}(f, \boxed{x}) & = & 0 \\ \\ \mathit{weight}(f, \overset{\wedge}{t_1\ t_2}) & = & (\mathit{sum}(f, t_1)\ +\ \mathit{sum}(f, t_2))\ + \\ & & (\mathit{weight}(f, t_1)\ +\ \mathit{weight}(f, t_2)). \end{array}$$

Intuitively $\mathit{sum}(f, t_1)$ and $\mathit{sum}(f, t_2)$ compute the "cost" of adding the left edge and the right edge respectively. The fact that the notion of *weight* on binary trees corresponds to the notion of weight on codes is proved by the following theorem:

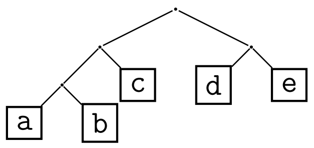
Theorem 2.3.2 (*weight Compute*) *If t is a binary tree with distinct leaves and forall a, $f(a)$ gives the frequency of the character a in the message m, then $\mathit{weight}(m, \mathit{computeCode}(t)) = \mathit{weight}(f, t)$.*

It proves that a binary tree of minimal weight whose leaves are the different characters of the text leads to an optimal code.

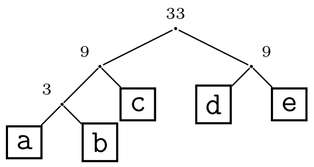
Before explaining how the formal proof proceeds, it is interesting to present the informal proof. The informal proof uses a more direct notion of weight that is:

$$\mathit{weight}(f, t) = \sum_{\boxed{x} \in t} f(x) \times \mathit{height}(\boxed{x})$$

where the function *height* returns the height of a subtree in a tree, i.e. the distance of the subtree from the root of the tree. To compute the weight of the tree one simply adds, for all leaves of the tree, the product of the frequency of the leave by its height. To give an example, consider the following tree:



with $f(a) = 1$, $f(b) = 2$, $f(c) = 3$, $f(d) = 4$, $f(e) = 5$. We can either compute the weight recursively:

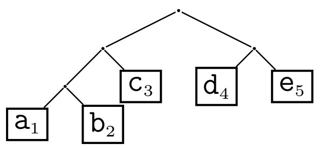


or use the summation formula with $height(a) = 3$, $height(b) = 3$, $height(c) = 2$, $height(d) = 2$, $height(e) = 2$ thus yielding:

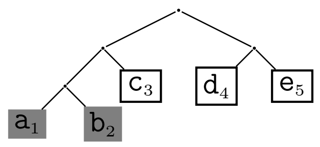
$$weight(f, t) = (1 \times 3) + (2 \times 3) + (3 \times 2) + (4 \times 2) + (5 \times 2) = 33.$$

The informal proof works as follows. It considers a binary tree of minimal weight and looks at a node of maximum height. As this node has maximum height, it is composed of two leaves. The frequencies of these two leaves should be minimal, otherwise given the summation formula exchanging these leaves with the two less frequent leaves would lead to a tree of smaller weight. Now, if we replace this node of maximum height by a new leaf whose frequency is the sum of the frequencies of the two previous leaves, the resulting tree should also be of minimal weight for the new alphabet. This is again proved by contradiction. If it were not the case, a tree of minimal weight for the new alphabet could be transformed into a tree of smaller weight for the original alphabet. Iterating the process gives us a way to check optimality.

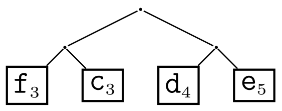
To give a concrete example, let us consider our previous example and annotate the leaves with their frequencies:



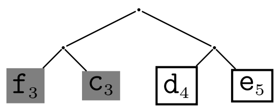
The tree has a node of maximum height whose leaves have minimal frequencies:



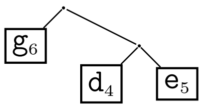
Replacing this node by a leaf `f` of frequency $1 + 2$ ($f(\texttt{f}) = 3$) yields:



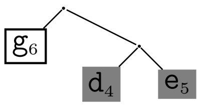
This tree has a node of maximum height whose leaves have minimal frequencies:



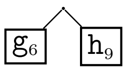
By creating a new leaf `g` with $f(\texttt{g}) = 3 + 3$, we get:



Again this tree has a node of maximum height whose leaves have minimal frequencies:



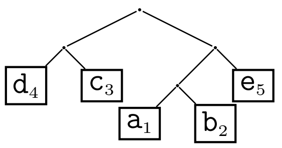
By creating a new leaf `h` with $f(\texttt{h}) = 4 + 5$, we get:



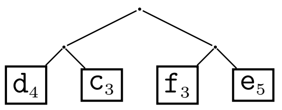
This tree has only one node, thus it is of minimal height. From this derivation, we can conclude that our initial tree was of minimal height. Note that the derivation follows exactly the same steps as in the execution of the example given in Section 1.

This informal proof has revealed to be difficult to formalise in a prover. There are several reasons:

- As described, the process does not work for all trees of minimal weight. For example, it does not work for the following tree that is of minimal weight:



After merging `a` and `b`, we get



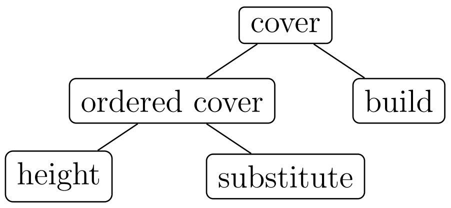
where the two nodes of maximum height do not have leaves of minimal frequencies. In fact, such a node only exists up to exchanging subtrees of same height (such exchanges do not modify the weight of the tree). Dealing formally with this kind of equivalence is always tedious.

- During the checking process new names are introduced. It is not clear how this mechanism could be formalised in a simple way.
- The process works in a bottom-up fashion. For top-down procedures, structural induction usually provides elegant proofs. For bottom-up ones, more sophisticated induction principles and larger invariants are often needed.
- The informal proof strongly relies on the notion of height of a subtree. Defining this notion formally is not so obvious. For example, the height can only be seen as a function if the tree is composed of distinct subtrees.

In the following, we explain how we have managed to overcome all these problems.

2.4 Covering

The central notion in our formalisation is the one of cover. From it, four other notions are derived:



Cover

A list of trees l is a *cover* of a tree t if t can be built from l using only nodes. More formally we have:

$cover([t], t).$

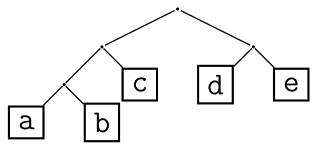
$cover(l, t) \quad \text{if} \quad l \approx [t_1;\ t_2] + l_1 \quad \text{and} \quad cover([\,{}_{t_1}{/\!\backslash}_{t_2}\,] + l_1, t).$

Two properties of *cover* are for example:

Theorem 2.4.1 (*Cover Perm*) *If we have* $l_1 \approx l_2$ *and* $cover(l_1, t)$*, then* $cover(l_2, t)$.

Theorem 2.4.2 (*Cover App*) *If we have* $cover(l_1, t_1)$ *and* $cover(l_2, t_2)$*, then* $cover(l_1 + l_2, {}_{t_1}{/\!\backslash}_{t_2})$.

What we want to find is a binary tree whose leaves are the different characters of the text and whose weight is minimal. Suppose that our text has five different characters `a`, `b`, `c`, `d`, `e`. Using the cover predicate, we can express this as finding a tree t of minimal weight such that $cover([\,\boxed{\texttt{a}};\ \boxed{\texttt{b}};\ \boxed{\texttt{c}};\ \boxed{\texttt{d}};\ \boxed{\texttt{e}}\,],\ t)$. For example, it is easy to check that $cover([\,\boxed{\texttt{a}};\ \boxed{\texttt{b}};\ \boxed{\texttt{c}};\ \boxed{\texttt{d}};\ \boxed{\texttt{e}}\,],\ t)$ holds for the following tree t:



Allowing a list of trees and not only a list of characters in the *cover* predicate simulates the mechanism of creating new elements of the alphabet. For example, in the previous tree when the leaves `a` and `b` are merged, what is searched for is a tree of minimal weight such that $cover([\,{}_{\boxed{\texttt{a}}}{/\!\backslash}_{\boxed{\texttt{b}}};\ \boxed{\texttt{c}};\ \boxed{\texttt{d}};\ \boxed{\texttt{e}}\,],\ t)$. There is no need anymore to create the new character `f`.

Build

To build trees of minimal weight we just need to slightly modify the definition of the *cover* predicate and require that the elements that are merged have minimal frequencies. This is done by the predicate *build*:

$$build(f, [t], t).$$

$$build(f, l, t)$$
$$\quad \text{if } l \approx [t_1;\ t_2] + l_1 \text{ and } \mathcal{O}(f, [t_1;\ t_2] + l_1) \text{ and } build(f, [\underset{t_1\ \ t_2}{/\backslash}] + l_1, t).$$

where the $\mathcal{O}$ predicate indicates that the list is ordered from the least frequent to the most frequent:

$$\mathcal{O}(f, []).$$

$$\mathcal{O}(f, [t]).$$

$$\mathcal{O}(f, [t_1;\ t_2] + l) \quad \text{if} \quad sum(f, t_1) \leq sum(f, t_2) \quad \text{and} \quad \mathcal{O}(f, [t_2] + l).$$

From these definitions, it follows that the *build* predicate is a refinement of the *cover* predicate:

Theorem 2.4.3 (*Build Cover*) *If $build(f, l, t)$, then $cover(l, t)$.*

Moreover, the weight of the tree that is built is independent of the way the list is ordered:

Theorem 2.4.4 (*Build Weight*) *If $build(f, l, t_1)$ and $build(f, l, t_2)$, then $weight(f, t_1) = weight(f, t_2)$.*

Of course what is left to be proved is that if we have $build(f, l, t)$ then t is of minimal weight for all trees t' such that $cover(l, t')$.

Ordered cover

In order to prove that a tree t such that $build(f, l, t)$ is of minimal weight, we need, as in the informal proof, to analyse the properties of trees of minimal weight. The first step in that direction is to refine the cover so to order the cover in a top-down left to right traversal. This is done by the *ordered cover*:

$$ordered\ cover([t], t).$$

$$ordered\ cover(l_1 + l_2,\ \underset{t_1\ \ t_2}{/\backslash})$$
$$\quad \text{if} \quad ordered\ cover(l_1, t_1) \quad \text{and} \quad ordered\ cover(l_2, t_2).$$

An ordered cover is a refinement of a cover and any cover can be ordered:

Theorem 2.4.5 (OrderedCover Cover) *If ordered cover$(l,\ t)$, then cover$(l,\ t)$.*

Theorem 2.4.6 (Cover OrderedCover) *If we have cover$(l,\ t)$, then there exists l_1 such that $l_1 \approx l$ and ordered cover$(l_1,\ t)$.*

Height

Now that we have ordered the cover, the next step is to introduce the height. As explained before, introducing the height as a function is not so simple, thus we do that using a predicate. We extend the notion of ordered cover with a predicate *height* with four arguments: an integer, a list of integers, a list of trees and a tree. The integer indicates the local height. The list of trees is an ordered cover of the tree. The list of integers corresponds to the height of the different trees of the cover with respect to the local height. Here is the definition:

$$\mathit{height}(n, [n], [t], t).$$

$$\mathit{height}(n, h_1 + h_2, l_1 + l_2, \overset{/\backslash}{t_1\ t_2})$$
$$\text{if} \quad \mathit{height}(n+1, h_1, l_1, t_1) \quad \text{and} \quad \mathit{height}(n+1, h_2, l_2, t_2).$$

This definition is a refinement of the one for ordered cover:

Theorem 2.4.7 (Height OrderedCover) *If height(n, h, l, t), then ordered cover(l, t).*

Furthermore it is always possible to find the height for an ordered cover:

Theorem 2.4.8 (OrderedCover Height) *If we have ordered cover(l, t), then for any n there exists a list h such that height(n, h, l, t).*

Substitute

The last transformation we need is the capability of exchanging subtrees. This is implemented by a predicate *substitute*:

$$\mathit{substitute}([t_1], [t_2], t_1, t_2).$$

$$\mathit{substitute}(l_1 + l_3, l_2 + l_4, \overset{/\backslash}{t_1\ t_3}, \overset{/\backslash}{t_2\ t_4})$$
$$\text{if} \quad \mathit{substitute}(l_1, l_2, t_1, t_2) \quad \text{and} \quad \mathit{substitute}(l_3, l_4, t_3, t_4).$$

The proposition $\mathit{substitute}(l_1, l_2, t_1, t_2)$ holds if the tree t_2 is the result of replacing in the tree t_1 all the trees in the list l_1 by the respective trees in the list l_2. Substitutions work on ordered covers:

Theorem 2.4.9 (*Subst Left OrderedCover*) *If substitute*(l_1, l_2, t_1, t_2)*, then ordered cover*(l_1, t_1)*.*

Theorem 2.4.10 (*Subst Right OrderedCover*) *If substitute*(l_1, l_2, t_1, t_2)*, then ordered cover*(l_2, t_2)*.*

Then substitution can be applied on lists of the same length:

Theorem 2.4.11 (*OrderedCover Subst*) *If length*(l_1) $=$ *length*(l_2) *and ordered cover*(l_1, t_1)*, then there exists a tree* t_2 *such that substitute*(l_1, l_2, t_1, t_2)*.*

2.5 Formal proof of Huffman's algorithm

The formal proof works as follows. It first shows that if we have $build(f, l, t)$, then t is of minimal weight for all tree t' such that $cover(l, t')$. It is proved by rule induction on $build(f, l, t)$: the proof by contradiction is transformed into a constructive proof.

The base case is trivial. For the inductive case, we consider $l \approx [t'_1;\ t'_2] + l_1$ with $\mathcal{O}(f, [t'_1;\ t'_2] + l_1$ and $build(f, [\begin{smallmatrix} & \wedge & \\ t'_1 & & t'_2 \end{smallmatrix}] + l_1, t)$. We take an arbitrary t' such that $cover(l, t')$ and want to prove that $weight(f, t) \leq weight(f, t')$. It is then sufficient to construct a t'' such that $weight(f, t'') \leq weight(f, t')$ and $cover([\begin{smallmatrix} & \wedge & \\ t'_1 & & t'_2 \end{smallmatrix}] + l_1,\ t'')$. Because of the induction hypothesis, we know that $weight(f, t) \leq weight(f, t'')$.

How do we construct this t''? We first know that there exists a list h such that $height(0, h, l, t')$. Now if we consider the product $\otimes_f$ between a list of integers and a list of trees defined as:

$$[n_1;\ n_2;\ \cdots;\ n_k] \ \otimes_f\ [t_1;\ t_2;\ \cdots;\ t_k] = \sum_{i=1}^{k} n_i \times weight(f, t_i)$$

the following property holds:

Theorem 2.5.1 (*Height Weight*) *If* $height(n, h, l, t)$*, then* $n \times sum(f, t) + weight(f, t) = h \otimes_f l$*.*

In the special case where $n = 0$ and $t = t'$, we have $weight(f, t') = h \otimes_f l$.

On the list h we consider the first occurrence of the maximum value that we call a. This first occurrence has some special properties.

Theorem 2.5.2 (*Height Disj*) *If $height(n, h_1 + [a] + h_2, l, t)$ and for all $b \in h_1$, $b < a$ and for all $c \in h_2$, $c \leq a$, then there exists h_3 such that $h_2 = [a] + h_3$ or we have $a = n$, $h_1 = h_2 = []$ and $l = [t]$.*

Theorem 2.5.3 (*Height Shrink*) *If $height(n, h_1 + [a;\ b] + h_2, l_1 + [t'_1;\ t'_2] + l_2, t)$ and for all $c \in h_1$, $c < a$ and for all $d \in [b] + h_2$, $d \leq a$, then $height(n, h_1 + [a - 1] + h_2, l_1 + [\overset{/\backslash}{t'_1\ t'_2}] + l_2, t)$.*

The first property tells us that the first maximum is always followed by another maximum. The second property asserts that these two maxima delimit a node of maximum height that can be merged. As we have

$$h = h_1 + [a;\ a] + h_2$$

we can decompose l in the same way

$$l = l_1 + [t_1;\ t_2] + l_2 \quad \text{where} \quad length(h_1) = length(l_1)$$

Now the following theorem

Theorem 2.5.4 (*$\otimes$ Reorder*) *If $length(h_1) = length(l_1)$ and $length(h_2) = length(l_2)$ and for all $b \in h_1, b \leq a$ and for all $c \in h_2, c \leq a$ and $l_1 + [t_1;\ t_2] + l_2 \approx [t'_1;\ t'_2] + l$ and $\mathcal{O}(f, [t'_1;\ t'_2] + l)$, then there exist l_3 and l_4 such that $length(l_1) = length(l_3)$ and $length(l_2) = length(l_4)$ and $[t'_1;\ t'_2] + l \approx l_3 + [t'_1;\ t'_2] + l_4$ and $(h_1 + [a;\ a] + h_2) \otimes_f (l_3 + [t'_1;\ t'_2] + l_4) \leq (h_1 + [a;\ a] + h_2) \otimes_f (l_1 + [t_1;\ t_2] + l_2)$.*

gives us a way to lower the product. If we have l_3 and l_4 of< Theorem 2.5.4 such that

$$l \approx l_3 + [t'_1;\ t'_2] + l_4$$

it is enough to take t'' such that $substitute(l, l_3 + [t'_1;\ t'_2] + l_4, t', t'')$.

What we have done is to exactly follow the informal proof. We took the first maximum in the list of heights. This maximum indicates a node of maximum height. This is given by Theorem 2.5.2 and Theorem 2.5.3. Then, Theorem 2.5.4 ensures us that putting there the two subtrees of minimum frequencies t'_1 and t'_2 always lowers the product.

The last step of the formal proof is to show that the program given in Appendix A computes effectively an optimal code. For this, we call $c2t$ the function that transforms a code into a binary tree

$$c2t(c) = toTree(buildTree(c)).$$

If we consider the result c of the application *(iter l)* and write l as $[(n_1, c_1);\ (n_2, c_2); \cdots;\ (n_k, c_k)]$, the invariant of the *iter* function is:

$$n_i = \mathit{sum}(f, \mathit{c2t}(c_i)) \qquad \text{for } i \leq k$$

and

$$\mathit{build}(f, [\mathit{c2t}(c_1);\ \mathit{c2t}(c_2); \cdots;\ \mathit{c2t}(c_k)], \mathit{c2t}(c)).$$

It is easily proved by induction. Since the tree in the *build* predicate is optimal, c is optimal.

3 Conclusions and Future Work

Formalising Huffman's algorithm has been a very interesting exercise. First of all, it shows how crucial it is to be able to change representations in a proof system. The initial problem of finding an optimal code has been reduced to a tree problem. Works like [4] indicate that more support could be provided to automate this aspect of the formalisation. Second, getting a formal proof that is relatively close to the informal proof was not so easy since the informal proof contains some technical difficulties. We were interested in the algorithm, so we had to turn the usual proof by contradiction into a constructive proof. It was not a surprise.

The first key step in the formalisation has been the introduction of the notion of *cover*. It gave us the induction principle. Because the *cover* definition was working in a bottom-up fashion, not much could be gained from the inductive hypothesis.

The second key step has been to turn the bottom-up definition into a top-down one, passing from the *cover* predicate to the *ordered cover* one. We believe that this transformation from bottom-up to top-down is a general technique. Interesting enough, the first version of the *ordered cover* predicate was using a more PROLOG-like definition using difference list [5] instead of concatenation. We thought that having functions in the conclusion of the predicate definitions could be a problem. It was not the case.

The last key step has been to realize that the first maximum in the list of heights was giving the position of a node of maximum height. It is again a consequence of the constructive flavour of our formalisation. In the informal proof, one takes an arbitrary node of maximum length. In our constructive proof we have to explicitly indicate which one to take.

In the formal proof, we have been using inductively defined predicates intensively. It is not clear whether this is only a question of personal style. Anyway it allows us to easily overcome most of the problems in mechanising the informal proof. A consequence of having so many inductively defined predicates is that rule induction has been a fundamental proof tool for our development.

Most proofs of the development are straightforward. There are mainly three technical proofs that correspond to Theorem 2.5.2, Theorem 2.5.3 and Theorem 2.5.4. The first two proofs are using rule induction. The last one is by case analysis and requires a fair amount of properties about list decompositions. The complete development is available at:
`ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Huffman/index.html`.

This work is an initial step towards formalising more elaborated compression/decompression algorithms. For example, an interesting challenge would be to get a certified version of a JPEG [6] decoder. It is in that direction that we plan to further work.

References

[1] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.

[2] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of IRE*, pages 1098–1101, 1952.

[3] Xavier Leroy. Objective Caml. Available at `http://pauillac.inria.fr/ocaml/`.

[4] Nicolas Magaud. Changing Data Representation within the Coq System. In *TPHOLs'03*, volume 2758 of *LNCS*, 2003.

[5] Leon Sterling and Ehud Shapiro. *The Art of Prolog: advanced programming techniques*. MIT Press, 1986.

[6] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, 1991.

A Huffman's algorithm in Ocaml

```
open List

type code = (char * bool list) list

let frequency_list l =
  let rec add a l = match l with
    [] -> [a, 1]
  | (b, n) :: l1 ->
     if (a = b) then (b, n + 1) :: l1 else (b, n) :: (add a l1)
  in fold_right add l []

let insert (n, a) l =
  let rec iter l =  match l with
    [] -> [n, a]
  | (m, b) :: l1 ->
     if (n <= m) then (n, a) :: l else (m, b) :: (iter l1)
  in iter l

let huffman m =
  let addl b l = map (fun (x, y) -> x, b :: y) l in
  let rec iter l = match l with
    | []     -> ([]: code)
    | [n, l] -> l
    | (n1, l1) :: (n2, l2) :: l ->
       iter
        (insert
            (n1 + n2, append (addl false l1) (addl true l2))
            l)
  in
    iter (fold_right (fun (a, n) l -> insert (n, [a, []]) l)
                     (frequency_list m)
                     [])
```