Formalization Description of Huffman Coding Trees Using Mizar

Takaya Ido[1], **Hiroyuki Okazaki**[1], **and Yasunari Shidama**[1]
[1]Shinshu University, 4-17-1 Wakasato Nagano-city, Nagano 380-8553, Japan

Abstract— *Mizar is a type of system known as a "proof checker," which automatically inspects the validity of formal mathematical proofs. Mizar, designed for computational descriptions of mathematics, was developed by Professor A. Trybulec et al. at the University of Bialystok, in Poland [1]. Various theorems can be formulated using the Mizar programming language, and their validity is automatically checked by Mizar's proof checker. The Mizar system contains a library known as the Mizar Mathematical Library, a repository of many formally described theorems and definitions whose validity has been already inspected, from which various applications can be sourced. In this report, we examine the future direction of formal definitions of source coding using Mizar, and as a specific example, we report on the formal description of Huffman coding [2].*

Keywords: Formal Verification, Mizar, Huffman trees

1. Formalization of tree structures

The so-called tree_* series in the Mizar Mathematical library (code abridged), known as a tree structure in Mizar, is expressed as a set of finite sequences of natural numbers. For example the set of finite sequences

$$\{\{\}, <*0*>, <*1*>, <*1,0*>, <*1,1*>\} \quad (1)$$

represents the tree shown in Figure 1. The top (or root) of

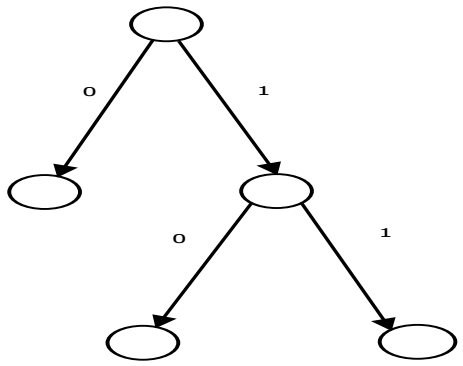

Fig. 1: Tree

the tree is represented by "," and a number is assigned to each branch. The terminating leaves and intermediate nodes are represented by the number sequence of their preceding branches

<*0*>,<*1*>,<*1,0*>,<*1,1*> (2)

In general, a tree is a set X of finite sequences of natural numbers that satisfy conditions (1) and (2) mentioned below.

1.1 Conditions of a tree

Consider a finite sequence of natural numbers belonging to set X, of arbitrary length m,

$$q =< *q_1, q_2, \cdots q_m* > \quad (3)$$

Condition 1: For an arbitrary natural number n, where n $\leq$ m, the subsequence

$$p =< *q_1, q_2, \cdots q_n* > \quad (4)$$

including the elements from the leading element of q to the nth element also belongs to set X. Condition 2: If a finite sequence of length m + 1, obtained by adding 1 to the end of q,

$$q^\frown < *l* >=< *q_1, q_2, \cdots q_m, l* > \quad (5)$$

belongs to set X, the finite sequence of length m + 1 obtained by adding an arbitrary natural number k (where k $\leq$ 1),

$$q^\frown < *k* >=< *q_1, q_2, \cdots q_m, k* > \quad (6)$$

also belongs to set X. The above-mentioned conditions (1) and (2) can be formalized as follows. The predicate specifying whether p is the leading element to the nth element (n $\leq$ m) extracted from the finite sequence q is

```
notation
  let p,q be FinSequence;
  synonym p is_a_prefix_of q for p c= q;
end;

definition
  let p,q be FinSequence;
  redefine pred p is_a_prefix_of q means
:: TREES_1:def 1
  ex n st p = q|Seg n;
end;
```

Here, Seg n is the set of natural numbers N $\neq$ 0 and N $\leq$ n .

```
definition
  let n be Nat;
  func Seg n -> set equals
```

```
:: FINSEQ_1:def 1
  { k where k is Nat: 1 <= k & k <= n };
end;
```

A proper subsequence is defined by

```
notation
 let p,q be FinSequence;
 synonym p is_a_proper_prefix_of
                  q for p c < q;
end;
```

The set of this proper subsequence is defined by

```
definition
  let p be FinSequence;
  func ProperPrefixes p -> set means
:: TREES_1:def 2
  for x being element holds
  x in it iff ex q being FinSequence
  st x = q & q is_a_proper_prefix_of p;
end;
```

In terms of the above definitions, conditions (1) and (2) are expressed as attributes (attr) of set X and also as a variable type definition as follows:

```
definition
  let X;
  attr X is Tree-like means
:: TREES_1:def 3
  X c= NAT* & (for p st p in
  X holds ProperPrefixes p c= X) &
  for p,k,n st p^<*k*>
  in X & n <= k holds
  p^<*n*> in X;
end;
definition
 mode Tree is Tree-like non empty set;
end;
```

In the tree shown in Figure 1, "" corresponds to the top of the tree. The nodes descending left and right from the top are denoted as " $<$ *1* $>$ " and " $<$ *0* $>$," respectively. Continuing this pattern, the nodes descending left and right from the preceding right node on the right are denoted as "$<$ *1, 0* $>$" and "$<$ *1, 1* $>$," respectively.

2. Root nodes and empty sequences

An empty finite sequence expressed as "" or "$<$ * $>$" NAT represents the root node at the top of the tree structure. By definition, this node constitutes an element of the tree and is formalized in the following proposition.

```
reserve T,T1 for Tree;
theorem :: TREES_1:22
  {} in T & <*> NAT in T;
```

Furthermore, an empty set of finite sequences also satisfies the conditions of a tree.

```
theorem :: TREES_1:23
  { {} } is Tree;
```

2.1 Sum and intersection of two trees

Because the sets of two trees comprise finite sequences of natural numbers, their intersection and sum also comprise finite sequences of natural numbers; thus, both operations generate trees. A formalized description of this statement is

```
reserve T,T1 for Tree;
registration
  let T,T1;
  cluster T \/ T1
  -> Tree-like;
  cluster T /\ T1
  -> Tree-like non empty;
end;
theorem :: TREES_1:24
  T \/ T1 is Tree;
theorem :: TREES_1:25
  T /\ T1 is Tree;
```

2.2 Finite trees

If tree T comprises a finite set of sequences, then it is said to be finite. The sum and intersection of two finite trees are also finite.

```
reserve fT,fT1 for finite Tree;
theorem :: TREES_1:26
  fT \/ fT1 is finite Tree;
theorem :: TREES_1:27
  fT /\ T is finite Tree;
```

2.3 Elementary trees

For an arbitrary natural number n, the set $<$ *k* $>$, which consists of natural numbers k, where k $<$ n, is called the elementary tree of n.

```
definition
  let n;
  func elementary_tree n -> Tree equals
:: TREES_1:def 4
  { <*k*> : k < n } \/ { {} };
end;
```

The elementary tree is a finite tree.

```
registration
  let n;
  cluster elementary_tree n -> finite;
end;
```

Also, given that $<$ *k* $>$, where k $<$ n, is an element of elementary_tree n, the following proposition can be established:

```
theorem :: TREES_1:28
 k < n implies <*k*>
 in elementary_tree n;
theorem :: TREES_1:29
 elementary_tree 0 = { {} };
theorem :: TREES_1:30
 p in elementary_tree n implies p =
    {} or ex k st k < n & p = <*k*>;
```

2.4 Leaf nodes

Because the leaf nodes lie at the bottom level of the tree structure, they possess no child nodes. If the finite sequence p is a leaf node of tree T, then the finite sequence q-an element of T-can never contain p as a proper subsequence. The formal definition is

```
definition
  let T;
  func Leaves T -> Subset of T means
:: TREES_1:def 5
  p in it iff p in T & not ex q st q in
          T & p is_a_proper_prefix_of q;
end;
```

For the tree shown in Figure 1,

$$\{\{\}, <*0*>, <*1*>, <*1,0*>, <*1,1*>\} \quad (7)$$

the leaves are

$$\{<*1,0*>, <*1,1*>\} \quad (8)$$

The set of all leaf nodes is defined as a variable, as shown below.

```
definition
  let T;
  assume
 Leaves T <> {};
  mode Leaf of T -> Element of T means
:: TREES_1:def 7
  it in Leaves T;
end;
```

2.5 Subtrees

If T is a tree and a finite sequence p is an element of T, then a finite sequence q may be connected to the end of p to create the finite sequence pq̂. Note that pq̂ (expressed as T|p) is also an element of T, and q is also a tree.

```
definition
  let T;
  let p such that
 p in T;
  func T|p -> Tree means
:: TREES_1:def 6  ::
  q in it iff p^q in T;
end;
```

For the tree shown in Figure 1,

$$T = \{\{\}, <*0*>, <*1*>, <*1,0*>, <*1,1*>\} \quad (9)$$

if p = < *1* >, then,

$$\begin{aligned} p\hat{}\{\} &= <*1*> \\ p\hat{} <*0*> &= <*1,0*> \\ p\hat{} <*1*> &= <*1,1*> \end{aligned} \quad (10)$$

so,

$$T|p = \{\{\}, <*0*>, <*1*>\}. \quad (11)$$

Therefore, T|p is a subtree of T as shown in Figure 2. From the above analysis, a subtree can be defined using

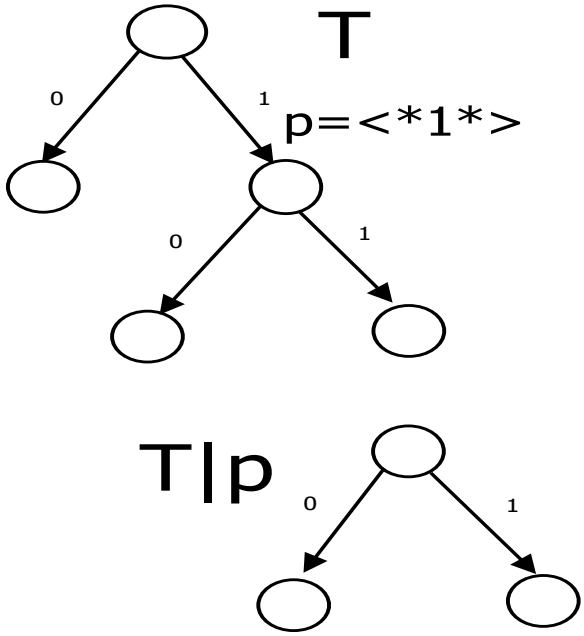

Fig. 2: Subtrees

the following variable type:

```
definition
  let T;
  mode Subtree of T -> Tree means
:: TREES_1:def 8
    ex p being Element of T st it = T|p;
end;
```

3. Huffman trees

Huffman coding is a type of entropy encoding from which an optimum code may always be constructed. A binary Huffman tree is based on the frequency of information symbols. The leaves of this tree correspond to information symbols, and the binary sequences denoting the paths from root to leaves are the code language. Here we outline a Huffman tree construction method.

3.1 Construction method for a Huffman tree

Step 1 : Every information symbol corresponds to a tree with a single node; the value of the node is the appearance frequency of the symbol. This step generates a set of trees in which all information symbols are either a root or leaf.

Step 2 : From the set of trees, select two nodes n1 and n2 with roots of least value. If multiple roots have the second lowest value, an arbitrary selection is made.

Step 3 : A new node n ′with its value equal to the sum of the values of nodes n1 and n2 is created. The resulting binary tree n" comprises a parent node n’ and child nodes n1 and n2.

Step 4 : Add n" to the set of trees.

Step 5 : If more than one tree exists, return to Step 2.

Step 6 : From each node of the constructed Huffman tree, the left and right branch is assigned 0 or 1 (hence, the path from root to leaf is described symbolically); whether the left or right branch is assigned 0 or 1 is arbitrary.

The arbitrariness of selection in Steps 2 and 6 means that, given the same information source, the Huffman tree is not uniquely constructed.

3.2 Formalization description of the Huffman tree structure

Source encoding is a mapping from the information source to the code space. In Mizar, mapping can be defined in two ways; using either a functor (func) or function (Function). The former is reserved in Mizar as a definition of a "term." Because Mizar is based on set theory, all terms are treated as "sets" and are referred to as functors. On the other hand, functions can also be treated as a set, with a family of functions constituting a new data type, "mode." As stated in Section 3.1 , the construction method for Huffman trees is not uniquely determined. Thus, in this report, a family of Huffman trees is defined as a "mode" of functions. The encoder is similarly defined in terms of functions or families of functions. Furthermore (although this is outside the scope of the present report), we consider that larger classes of code, such as entropy encoding and compact code, can be formalized in a manner similar to the Huffman code in Mizar.

In this section, a formal description of the construction method outlined in Section 2.1 is presented. We consider a binary tree with values (binary DecoratedTree) described in Section 1. A Huffman tree is defined as a binary tree i.e., a finite set of binary sequences, with the binary tree function as its domain and the direct products of sources and their occurrence probabilities as its range. Because a binary tree is a code space, the function is a decoder, and a one-to-one mapping exists between them. Therefore, the definition for encoding should be derivable from the binary tree definition. First, we define the predicate that must be satisfied by the Huffman tree construction method.

```
definition
let SOURCE be non empty finite set;
let p be Probability of
       (Trivial-SigmaField SOURCE);
let Tseq be  FinSequence
        f BoolBinFinTrees ExtSOURCE ;
let q being FinSequence of NAT;
pred Tseq,q,SOURCE,
p is_constructingHuffman
 means
:: HUFFMAN:def 12
Tseq.1 = Initial-Trees(SOURCE,p)
 & len Tseq = card (SOURCE)
 & ( for i be Nat st 1<= i
 & i < len Tseq
  holds
  ex X,Y  be non empty finite
          Subset of BinFinTrees
     ExtSOURCE
  st
  ex s,t,v be  finite  binary
    DecoratedTree of ExtSOURCE
  st
  Tseq.i = X
  & s is_MinValueTree_of X
  & Y = X \ {s}
  & t is_MinValueTree_of Y
  & v in
   {MakeTree (t,s,MaxVl(X) + 1),
    MakeTree (s,t,MaxVl(X) + 1) }
  & Tseq.(i+1) = (X \ {t,s}) \/ {v})
  & ( ex T be finite  binary
  DecoratedTree of ExtSOURCE
      st {  T  } = Tseq.(len Tseq) )
  & dom q = Seg  card (SOURCE)
  & (for k be Nat st k
     in Seg card (SOURCE)
     holds q.k = card(Tseq.k)
     & q.k <> 0 )
  & (for k be Nat holds
    (k  < card (SOURCE)  implies
     q.(k+1) = q.1 - k))
  & (for k be Nat
     st 1<=k & k < card (SOURCE)
     holds  2 <= q.k );
end;
definition
let SOURCE be non empty finite set;
let p be Probability of
       (Trivial-SigmaField SOURCE);
```

```
let T be  finite  binary
       DecoratedTree of ExtSOURCE ;
pred T,p,SOURCE is_HuffmanCode-Like
 means
:: HUFFMAN:def 13
ex  Tseq be  FinSequence
      of BoolBinFinTrees ExtSOURCE,
    q being FinSequence of NAT
st Tseq,q,SOURCE,
p is_constructingHuffman
   & {T} = Tseq.( (len Tseq) ) ;
end;
```

The following is an explanatory outline. In HUFFMAN:def 12 from above,

1) Step 1 is the processing step. In the initial set of trees, all information symbols correspond to trees with a single node, and the node values are the frequencies of the information symbols. Steps 2 to 4 are iterative steps. Two trees are selected from the set of trees in an intermediate process and combined into a new tree. The selected trees are then removed, and the new tree is added to the tree set. This serial process is represented formally by introducing the finite sequence of the set of binary trees (Tseq) with values in ExtSOURCE.
2) For the set of given information sources SOURCE and its appearance probability distribution p, the following proposition guarantees that Tseq (the finite sequence of the set of binary trees) exists.

```
theorem :: HUFFMAN:3
for  SOURCE be non empty finite set,
     p be Probability of
     (Trivial-SigmaField SOURCE)
     st 2 <= card (SOURCE)
holds
ex  Tseq be  FinSequence
  of BoolBinFinTrees ExtSOURCE,
  q being FinSequence of NAT
st
  Tseq,q,SOURCE,
  p is_constructingHuffman;
```

3) ExtSOURCE is the set of all pairings between the numbers (natural numbers) attached to the nodes of the sequentially generated Huffman trees and their appearance probabilities (real numbers) given by probability p.

```
definition
func  ExtSOURCE  -> non empty set
  equals
:: HUFFMAN:def 2
  [:NAT,REAL:] ;
end;
```

By formalizing a Huffman tree as a binary tree with a value in ExtSOURCE, a unique pairing of number and appearance probability can be mapped to each node.

4) The initial tree set is the initial value of Tseq, (Tseq.1), defined as Initial-Trees(SOURCE,p). The iterations proceed through Tseq.i to process the ith set of trees. The final iteration of Tseq is expressed as Tseq.(lenTseq) (determined by the length lenTseq of Tseq). Tseq.(lenTseq) comprises a single Huffman tree expressed as

```
{T} = Tseq.( (len Tseq) ) ;
```

5) The set of initial trees Initial-Trees(SOURCE,p) is a set of trees each with a single node, in which appearance probability p.x, given by the information number x and its probability, is mapped to an elementary tree elementarytree0 = (as described in Section 1),

elementary tree0 - - > [(canFSSOURCE)."x,p.x] and is defined as follows:

```
definition
let SOURCE be non empty finite set;
let p be Probability
    of (Trivial-SigmaField SOURCE);
func  Initial-Trees(SOURCE,p)
-> non empty finite Subset of
            BinFinTrees ExtSOURCE
equals
:: HUFFMAN:def 5
{T where T is Element of
     FinTrees ExtSOURCE :
  T is finite  binary
    DecoratedTree of ExtSOURCE
  & ex x be Element of SOURCE st
    T =  elementary_tree 0 -->
    [(canFS SOURCE)".x , p.{x}]} ;
end;
```

Here,(canFS SOURCE)" is a mapping that uniquely corresponds natural numbers to elements x from the set of information sources SOURCE.

Among the set of trees in the ith iteration of Tseq.i, the two trees rooted by the lowest appearance probability (the second coordinate of their attached value) are selected (according to the predicate "is MinValueTree of"). The formal procedure by which s and t are removed from Tseq.1 and the synthesized tree is added to construct Tseq.(i + 1) is shown below. This formulation includes Step 3.

```
Tseq.i = X
& s is_MinValueTree_of X
& Y = X \ {s}
& t is_MinValueTree_of Y
& v in {MakeTree (t,s,MaxVl(X) + 1),
    MakeTree (s,t,MaxVl(X) + 1) }
& Tseq.(i+1) = (X \ {t,s} ) \/ {v})
```

The action of fetching the first and second

coordinates of the value corresponding to the binary tree, which has a value in ExtSOURCE, is described below.

```
definition
let p be DecoratedTree of ExtSOURCE;
func  Vrootr p -> Real
 equals
:: HUFFMAN:def 6
    (p.{}) ‘2 ;
end;

definition
let p be DecoratedTree of ExtSOURCE;
func  Vrootl p -> Nat
 equals
:: HUFFMAN:def 7
    (p.{}) ‘1 ;
end;
```

The predicate "is MinValueTree of" is hence formalized as

```
definition
let X be non empty finite Subset of
      BinFinTrees ExtSOURCE;
let p be   finite binary
      DecoratedTree of ExtSOURCE ;
pred  p  is_MinValueTree_of X
means
:: HUFFMAN:def 10
p in X & for q be finite binary
      DecoratedTree of ExtSOURCE
st
q in X holds (Vrootr p) <= Vrootr q;
end;
```

and MakeTree is defined as follows.

```
definition
let p,q be finite  binary
    DecoratedTree of ExtSOURCE;
let k be Nat;
func MakeTree (p,q,k)
   -> finite  binary
    DecoratedTree of ExtSOURCE
 equals
:: HUFFMAN:def 9
 [k,(Vrootr p) +(Vrootr q)]
 -tree (p,q);
end;
```

MakeTree (t, s, MaxVl(X) + 1) generates a new tree from s and t, as described in Section 1. The maximum value MaxVI(X) of all numbers attached to the roots of the trees belonging to X = Tseq.i is increased by 1 and paired with the sum of appearance probabilities of the roots of s and t. This pairing then corresponds to the root of the newly synthesized tree.

Maximum value MaxVl is formalized as shown below. The second coordinate of the value corresponding to the root of tree p, belonging to the set T of binary trees with values in ExtSOURCE, is defined as

```
definition
let T be  finite  binary
      DecoratedTree of ExtSOURCE;
let p be Element of (dom T) ;
func  Vtree (T,p) -> Real
 equals
:: HUFFMAN:def 8
    (T.p) ‘2 ;
end;
```

by which the number assigned to the root of the tree in X = Tseq.i is fetched.

Next, we describe how the largest of these numbers is fetched.

```
definition
let X be non empty finite Subset of
        BinFinTrees ExtSOURCE;
func MaxVl(X)  -> Nat
 means
:: HUFFMAN:def 11
 ex L be non empty
 finite Subset of NAT
 st L = {Vrootl p where p
 is Element of
 BinFinTrees ExtSOURCE: p in X }
 & it = max L ;
end;
```

The processing of Step 6 can be encompassed in a definition that clearly expresses (including the existence of the mapping itself) the mapping that corresponds each node of the processed Huffman tree to a finite sequence of 0,1.

```
definition
let SOURCE be non empty finite set;
let p be Probability of
       (Trivial-SigmaField SOURCE);
mode entropyCode-encoder of SOURCE,p
    -> Function of SOURCE, BOOLEAN*
   means
:: HUFFMAN:def 19
it is one-to-one
& ex T be  finite  binary
       DecoratedTree of ExtSOURCE
 st
sT,p,SOURCE is_HuffmanCode-Like
& rng it = Leaves (dom T)
& for x be Element of SOURCE
  holds
  T.(it.x)
```

```
  = [(canFS(SOURCE))".x ,p.{x}];
end;
```

The appearance probability of each node that is not a leaf of the final Huffman tree is the sum of the appearance probabilities of each of its child nodes. This is expressed in the following proposition:

```
theorem :: HUFFMAN:19
for SOURCE be non empty finite set,
 p be Probability of
      (Trivial-SigmaField SOURCE),
 T be  finite  binary
      DecoratedTree of ExtSOURCE
st  T,p,SOURCE is_HuffmanCode-Like
holds
 for  t,s,r be Element of dom T
  st
  t in ( dom T \ (Leaves (dom T)) )
  & s = (t^<* 0 *> )
  & r = (t^<* 1 *> )
  holds
   Vtree (T,t) =
     Vtree (T,s) + Vtree (T,r);
```

To prove this proposition, we must show that a similar proposition holds for the trees in Tseq.i, the set of trees in the ith iteration. This is achieved by mathematical induction involving i, as follows.

```
theorem :: HUFFMAN:18
for SOURCE be non empty finite set,
   p be Probability of
      (Trivial-SigmaField SOURCE),
   Tseq be  FinSequence
      of BoolBinFinTrees ExtSOURCE,
   q being FinSequence of NAT
 st  Tseq,q,SOURCE,
     p is_constructingHuffman
holds
 for i be Nat st 1 <=i
     & i <=len Tseq
   holds
    for  T be  finite  binary
      DecoratedTree of ExtSOURCE
    for  t,s,r be Element of dom T
     st T in Tseq.i & t in
       ( dom T \ (Leaves (dom T)) )
      & s = (t^<* 0 *> )
      & r = (t^<* 1 *> )
     holds
     Vtree (T,t)
     = Vtree (T,s) + Vtree (T,r);
```

4. Closing remarks

In this report, we propose a series of formal definitions for source coding in Mizar. Specifically, we reported a formal definition of Huffman coding. First, we defined a method of constructing a Huffman tree. From this definition, we defined the encoder connecting the information source to the code space and the coding scheme. Proofs of definitions and theorems relating to the Huffman code were formulated in Mizar and verified using Mizar's checker. Future work will address proofs on the characteristics of Huffman encoding, such as length of code, the sibling property [3], and optimality.

Acknowledgment

This study was partly supported by JSPS KAKENHI 21240001 and 22300285.

References

[1] Mizar System: Available at http://mizar.org/.
[2] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the I.R.E. September, pp.1098-1102, 1952.
[3] R.G. Gallager, Variations on a Theme by Huffman, IEEE Transactions on Information Theory,Vol. IT- 24,NO. 6, pp. 668-674, 1978.