Algorithms in Functional Programming

Lars-Henrik Eriksson

Functional Programming 1

Original presentation by Tjark Weber based on notes by Pierre Flener, Jean-Noël Monette, Sven-Olof Nyström

Algorithms and Data Structures

Designing algorithms and data structures to solve problems lies at the heart of programming.

Functional programs often use similar algorithms (e.g, for sorting) and data structures (e.g., lists, trees, queues) as imperative programs.

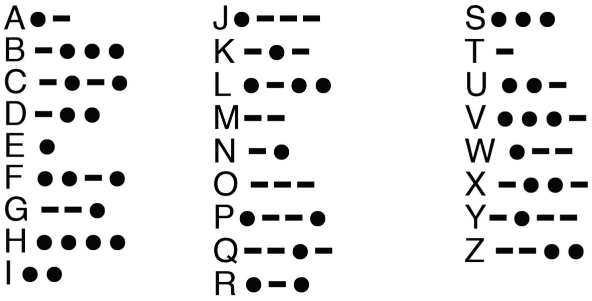
However, pattern matching, recursion, higher-order functions, and especially non-mutable data lead to a distinct functional flavor.

Today

Table of Contents

Morse Code

Morse code:



A	●▬	J	●▬▬▬	S	●●●
B	▬●●●	K	▬●▬	T	▬
C	▬●▬●	L	●▬●●	U	●●▬
D	▬●●	M	▬▬	V	●●●▬
E	●	N	▬●	W	●▬▬
F	●●▬●	O	▬▬▬	X	▬●●▬
G	▬▬●	P	●▬▬●	Y	▬●▬▬
H	●●●●	Q	▬▬●▬	Z	▬▬●●
I	●●	R	●▬●		

Note that more common letters (e.g., E, T) have shorter encodings.

Application: File Compression

- Characters are usually encoded using a fixed number of bits (e.g., 8).
- However, to reduce space requirements, we can use a **variable-length** encoding based on the frequency of characters, so that common characters have shorter encodings.
- One variable-length encoding is known as **Huffman coding**. It can be shown that Huffman coding is optimal, assuming that the frequency of each character is known.
- Huffman coding is a building block of many compression algorithms (including ZIP, GZIP, JPEG).
- Today, we will see how one can implement Huffman coding in SML.

Huffman Trees

A **Huffman tree** is a full binary tree such that

- each leaf is labelled with a character,[1]
- each subtree (i.e., each leaf and each node) is labelled with the frequency of the characters in that subtree,
- subtrees with larger frequencies are higher up in the tree.

[1]Huffman coding can similarly be used for other data, e.g., words.

Huffman Trees

A **Huffman tree** is a full binary tree such that

- each leaf is labelled with a character,[1]
- each subtree (i.e., each leaf and each node) is labelled with the frequency of the characters in that subtree,
- subtrees with larger frequencies are higher up in the tree.

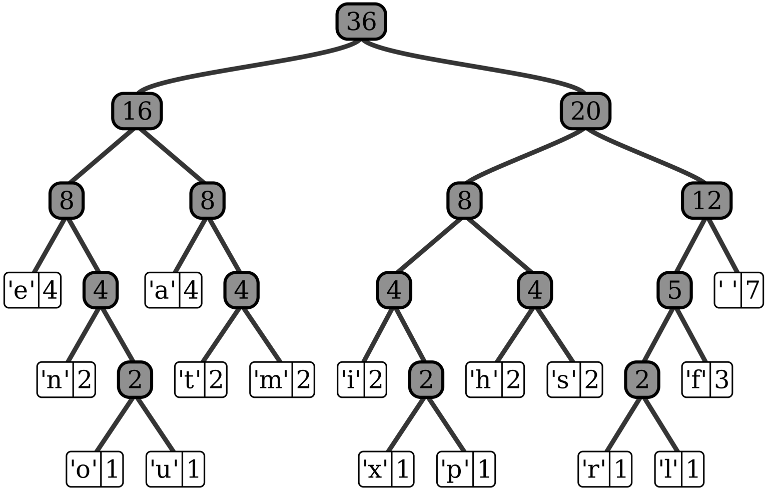
Huffman trees in SML:

```
datatype hufftree = Leaf of int * char
                  | Node of int * hufftree * hufftree
```

[1]Huffman coding can similarly be used for other data, e.g., words.

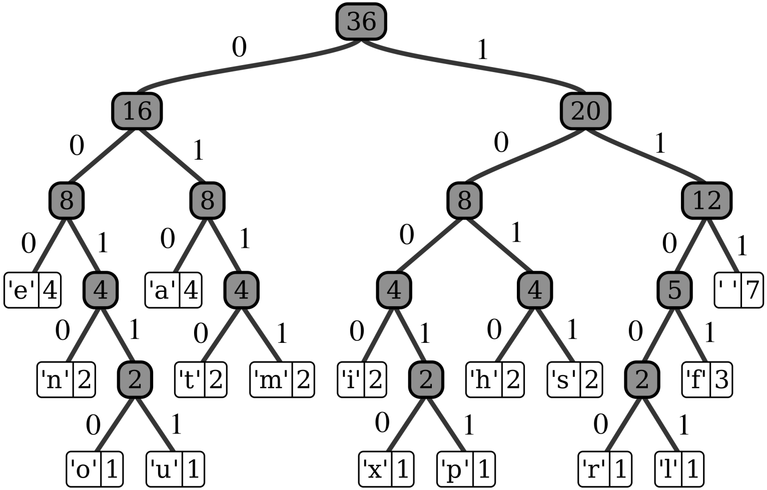
Huffman Trees: Example

A Huffman tree for the text “this is an example of a huffman tree”:



Huffman Trees: Character Encoding

A Huffman tree defines a binary code for each character:



Huffman Trees: Character Encoding (cont.)

The **Huffman code** of a character is the sequence of 0/1 labels from the root to the corresponding leaf.

For instance, in the example tree on the previous slide:

- `e` is encoded as 000
- `n` is encoded as 0010
- `o` is encoded as 00110
- etc.

The Huffman code is a **prefix code**: no valid code word is the prefix of another code word.

Huffman Coding: Main Operations

We need to implement four main operations:

1. Counting the frequency of each character in a text
2. Building a Huffman tree (from given character frequencies)
3. Encoding a text (using a given Huffman tree)
4. Decoding a sequence of bits (using a given Huffman tree)

We will represent the original text as a list of characters, and the encoded text as a list of integers $0/1$.

Table of Contents

Counting Character Frequencies

We want to compute a **map** (i.e., dictionary) from characters to their respective frequencies.

Recall our implementation of dictionaries (cf. last lecture):

```
signature DICTIONARY =
sig
  type (''a,'b) T
  val empty: (''a,'b) T
  val insert : ''a -> 'b -> (''a, 'b) T -> (''a, 'b) T
  val lookup: ''a -> (''a, 'b) T -> 'b option
  val list_of : (''a,'b) T -> (''a * 'b) list
end

structure Dictionary :> DICTIONARY = ...
```

Counting Character Frequencies (cont.)

1. Start with an empty dictionary.
2. For each character *c* in the text: if *c* was already in the dictionary, increment its frequency; otherwise, insert it with a frequency of 1.

```
(* frequencies cs
   TYPE: char list -> (char, int) Dictionary.T
   PRE: true
   POST: a dictionary that maps each char in cs to the number of its
     occurrences in cs
 *)
```

Counting Character Frequencies (cont.)

1. Start with an empty dictionary.
2. For each character *c* in the text: if *c* was already in the dictionary, increment its frequency; otherwise, insert it with a frequency of 1.

```
(* frequencies cs
   TYPE: char list -> (char, int) Dictionary.T
   PRE: true
   POST: a dictionary that maps each char in cs to the number of its
     occurrences in cs
 *)
fun frequencies cs =
  foldl (fn (char, dict) =>
    case Dictionary.lookup char dict of
      NONE => Dictionary.insert char 1 dict
    | SOME n => Dictionary.insert char (n+1) dict) Dictionary.empty cs
```

Table of Contents

Building a Huffman Tree

Let's build a Huffman tree from a given dictionary of character frequencies.

Idea:

1. Create a singleton tree for each character in the dictionary.
2. Insert all singleton trees into a collection.
3. While there is more than one tree in the collection:
    1. Remove the two trees with the smallest frequencies from the collection.
    2. Merge them into one tree whose frequency is the sum of the individual frequencies.
    3. Insert this new tree into the collection.

$\Rightarrow$ We need a *priority queue* (i.e., a collection with easy access to the least element).

Priority Queues: Interface

```
signature PRIORITY_QUEUE =
sig
  type 'a T
  val empty: ('a * 'a -> order) -> 'a T
  val insert : 'a -> 'a T -> 'a T
  val least : 'a T -> 'a * 'a T
  val is_empty : 'a T -> bool
  exception Empty
end
```

Priority Queues: Implementation

For simplicity, we use a list-based implementation of priority queues.
(There are other implementations with better complexity.)

```
structure Priority_Queue :> PRIORITY_QUEUE =
  struct
    datatype 'a T = PQ of ('a * 'a -> order) * 'a list
    fun empty order = PQ (order, [])
    fun insert x (PQ (order, qs)) = ...
    fun least (PQ (order, [])) = raise Empty
      | least (PQ (order, q::qs)) = (q, PQ (order, qs))
    fun is_empty (PQ (_, qs)) = null qs
    exception Empty
  end
```

Building a Huffman Tree: Implementation

```
(* hufftree dict
   TYPE: (char,int) Dictionary.T -> hufftree
   PRE: dict is non-empty
   POST: a Huffman tree based on the frequency data in dict
 *)
fun hufftree dict =
  let
    val trees = map (fn (c,n) => Leaf (n,c)) (Dictionary.list_of dict)
    fun hufftree_compare (t1, t2) =
      Int.compare (frequency t1, frequency t2)
    val queue = foldl (fn (t,q) => Priority_Queue.insert t q)
      (Priority_Queue.empty hufftree_compare) trees
    fun merge_hufftrees t1 t2 =
      Node (frequency t1 + frequency t2, t1, t2)
    ...
```

Building a Huffman Tree: Implementation (cont.)

```
  ...
  fun merge_queue q =
    let val (t1,q) = Priority_Queue.least q
    in
      if Priority_Queue.is_empty q then
        t1
      else
        let val (t2,q) = Priority_Queue.least q
        in
          merge_queue
            (Priority_Queue.insert (merge_hufftrees t1 t2) q)
        end
    end
in
  merge_queue queue
end
```

Table of Contents

Encoding a Text

Let's encode a list of characters, using a given Huffman tree.

Idea:

1. Traverse the tree to build a code dictionary (i.e., a mapping from characters to code words).
2. For each character in the text:
    1. Look up the corresponding code word in the code dictionary.
3. Concatenate all code words.

Building a Code Dictionary: Implementation

```
(* code_dict t
   TYPE: hufftree -> (char,int list ) Dictionary.T
   PRE: t is not a Leaf
   POST: a dictionary mapping characters in t to their Huffman
     encoding
 *)
```

Building a Code Dictionary: Implementation

```
(* code_dict t
   TYPE: hufftree -> (char,int list ) Dictionary.T
   PRE: t is not a Leaf
   POST: a dictionary mapping characters in t to their Huffman
     encoding
 *)
fun code_dict t =
  let
    fun code_dict' dict path (Leaf (_,c)) =
          Dictionary.insert c (rev path) dict
      | code_dict' dict path (Node (_,l,r)) =
          code_dict' (code_dict' dict (0::path) l) (1::path) r
  in
    code_dict' Dictionary.empty [] t
  end
```

Encoding a Text: Implementation

```
(* encode_with_tree t cs
   TYPE: hufftree -> char list -> int list
   PRE: t is not a Leaf and contains all characters in cs
   POST: the Huffman coding of cs under the given tree t
 *)
```

Encoding a Text: Implementation

```
(* encode_with_tree t cs
   TYPE: hufftree -> char list -> int list
   PRE: t is not a Leaf and contains all characters in cs
   POST: the Huffman coding of cs under the given tree t
 *)
fun encode_with_tree t cs =
  let
    val dict = code_dict t
  in
    List.concat
      (map (fn c => valOf (Dictionary.lookup c dict)) cs)
  end
```

Encoding a Text (cont.)

Let's encode a list of characters, using the tree that we obtain from their frequencies (returning both the tree and the encoding).

```
(* encode cs
   TYPE: char list -> hufftree * int list
   PRE: cs contains at least two distinct characters
   POST: (a Huffman tree based on the character frequencies in cs,
     the Huffman coding of cs under this tree)
 *)
```

Encoding a Text (cont.)

Let's encode a list of characters, using the tree that we obtain from their frequencies (returning both the tree and the encoding).

```
(* encode cs
   TYPE: char list -> hufftree * int list
   PRE: cs contains at least two distinct characters
   POST: (a Huffman tree based on the character frequencies in cs,
     the Huffman coding of cs under this tree)
 *)
fun encode cs =
  let
    val t = hufftree (frequencies cs)
  in
    (t, encode_with_tree t cs)
  end
```

Table of Contents

Decoding a Sequence of Bits

Let's decode a sequence of bits (i.e., a list of 0/1 integers), using a given Huffman tree.

Idea:

1. Start at the root of the Huffman tree.
2. For each bit in the encoding sequence:
    1. If it is a 0, go to the left subtree. If it is a 1, go to the right subtree.
    2. When we have reached a leaf, return the corresponding character. Go back to the root of the tree and decode the remaining bits.

Decoding: One Character at a Time

```
(* decode_one t xs
   TYPE: hufftree -> int list -> char * int list
   PRE: xs = w @ xs', where w is a valid Huffman code word for t
   POST: (the decoding of w under t, xs')
 *)
```

Decoding: One Character at a Time

```
(* decode_one t xs
   TYPE: hufftree -> int list -> char * int list
   PRE: xs = w @ xs', where w is a valid Huffman code word for t
   POST: (the decoding of w under t, xs')
 *)
fun decode_one (Leaf (_,c)) xs = (c, xs)
  | decode_one (Node (_,l,_)) (0::xs) = decode_one l xs
  | decode_one (Node (_,_,r)) (1::xs) = decode_one r xs
  | decode_one (Node _) _ = raise Domain
```

Decoding: Implementation

```
(* decode t xs
   TYPE: hufftree -> int list -> char list
   PRE: xs is a concatenation of valid Huffman code words for t
   POST: the decoding of xs under t
 *)
```

Decoding: Implementation

```
(* decode t xs
   TYPE: hufftree -> int list -> char list
   PRE: xs is a concatenation of valid Huffman code words for t
   POST: the decoding of xs under t
 *)
fun decode t [] =
      []
  | decode t xs =
      let
        val (c,xs) = decode_one t xs
      in
        c :: decode t xs
      end
```

Huffman Coding: Exercise

- What happens when we try to encode the empty list of characters?

```
encode [];
```

Huffman Coding: Exercise

- What happens when we try to encode the empty list of characters?

```
encode [];
```

- What happens when we try to encode (and then decode) a list that contains only one (distinct) character?

```
val (t,xs) = encode [#"x"];
decode t xs;

val (t,xs) = encode [#"x", #"x", #"x"];
decode t xs;
```

Huffman Coding: Exercise

- What happens when we try to encode the empty list of characters?

```
encode [];
```

- What happens when we try to encode (and then decode) a list that contains only one (distinct) character?

```
val (t,xs) = encode [#"x"];
decode t xs;

val (t,xs) = encode [#"x", #"x", #"x"];
decode t xs;
```

Exercise: Modify our implementation of Huffman coding so that these cases work as expected.

Table of Contents

Huffman Coding: Genericity

The code that we have written uses list-based dictionaries and priority queues.

However, any other implementation of dictionaries and/or priority queues could be used instead.

In SML, we can use a functor to write a **generic** version of Huffman coding.

- This will use (signatures for) dictionaries and priority queues.
- The actual implementation of these data types (i.e., the structures implementing them) can be specified later.

A Signature for the Parameters

First, let's write a signature that contains all parameters to our Huffman coding algorithm:

```
signature PARAMETERS =
sig
  structure D : DICTIONARY
  structure PQ : PRIORITY_QUEUE
end
```

Note that this signature only requires the *interface* for dictionaries and queues (i.e., the signatures DICTIONARY and PRIORITY_QUEUE). Structures that implement dictionaries and queues may be written later.

A Signature for the Result

Let's also write a signature for the structure that implements Huffman coding. (This is optional, but useful to hide auxiliary functions and other implementation details.)

```
signature HUFFMAN_SIG =
sig
  type hufftree
  val encode: char list -> hufftree * int list
  val decode: hufftree -> int list -> char list
end
```

A Functor for Huffman Coding

```
functor HUFFMAN_CODING(Parameters: PARAMETERS)
  : HUFFMAN_SIG =
struct

  (* the same code as earlier in this lecture, but now using
     Parameters.D instead of Dictionary (which was our list-based
     implementation of dictionaries) and Parameters.PQ instead of
     Priority_Queue *)
   ...

end
```

A Functor for Huffman Coding Applied

We can use the functor HUFFMAN_CODING to obtain an implementation of Huffman coding that uses list-based dictionaries and priority queues:

```
structure List_Parameters =
struct
  structure D = Dictionary
  structure PQ = Priority_Queue
end

structure List_Coding = HUFFMAN_CODING(List_Parameters)

List_Coding.encode
  (String.explode "this is an example of a huffman tree")
```