Data Encryption Standard

Example 3

CSE 791: E-3

Title: Data Encryption Standard
SML files: des.sml
Objective: Details of Data Encryption Standard

1 Introduction

The Data Encryption Standard (DES) algorithm is a secret or symmetric key algorithm. DES is a block cipher, that is it encrypts plaintext 64-bits at a time to produce 64-bits of ciphertext.

The basic idea behind DES is to uniformly distribute the effects of each bit of the key on the plaintext being encrypted. At each encryption stage, the left and right halves of the encrypted text are swapped, the key is rotated, and a substitution is performed based on some subset of a permutation of the key.

Figure 1 is a top-level block diagram of the DES algorithm.

The texts by Kaufman [1] and Schneier [2] focus on DES describing the permutations and substitutions in DES. Our development of DES will yield a formal theory of DES. Once we have a theory, we can give proofs showing how DES is used correctly. In addition, we will give a top-down executable description of DES in Standard ML (SML) of New Jersey 0.93.

2 Basic Properties and Functions

2.1 Properties of Exclusive-Or

DES (as well as other crypto algorithms) uses exclusive-or $\oplus$ extensively. $\oplus$ is its own inverse, i.e. given $x \oplus y$ and $y$ we can retrieve $x$. This is a crucial property for decryption.

A typical programming practice is to define $\oplus$ on bits instead of the booleans, where the set of bits \{0,1\} is understood to be a proper subset of the natural numbers. Our formalization (and SML programs) will adopt the same practice. We will use relativized quantifiers as described in Manna and Waldinger [3] to restrict our theorems and definitions to bits as needed.

Bit is the predicate which is true when applied to either 1 or 0.

$$\text{Bit } x = (x = 0) \lor (x = 1)$$

(1)
Figure 1: Overview of DES

Exclusive-or exor as defined on the booleans is:

\[ \text{exor}(x, y) = \neg x \land y \lor x \land \neg y \]  \hspace{1cm} (2)

intToBool maps 0 to false and 1 to true.

\[ \text{intToBool } x = (x = 0 \rightarrow \text{false} \mid \text{true}) \]  \hspace{1cm} (3)

boolToInt maps true to 1 and false to 0.

\[ \text{boolToInt } x = x \rightarrow 1 \mid 0 \]  \hspace{1cm} (4)

\( \oplus \) is defined as the Bit version of exor.

\[ x \oplus y = \text{boolToInt}(\text{exor}(\text{intToBool } x, \text{intToBool } y)) \]  \hspace{1cm} (5)
Based on the above definitions the following theorems are easily proved by simple case analysis or equational substitution.

\[
\begin{align*}
(\forall \text{Bit } x).x \oplus x &= 0 \\
(\forall \text{Bit } x).x \oplus 0 &= x \\
(\forall \text{Bit } x\ y).x \oplus y &= y \oplus x \\
(\forall \text{Bit } x\ y\ z).(x \oplus y) \oplus z &= x \oplus (y \oplus z)
\end{align*}
\]

We can prove that \(\oplus\) is its own inverse.

\[
(\forall \text{Bit } x\ y).[x = (x \oplus y) \oplus y]
\]

The proof restricts the domain of \(\oplus\) to \text{Bit}. This is appropriate not only for theoretical reasons of correctness, but because some implementations of \(x \oplus y\) use the definition \(x \neq y\) which clearly has a domain beyond \text{Bit}.

### 2.2 MAP2

DES, as well as other crypto functions, maps bit-operations over words. A convenient representation for words is a list. Lists are sequences of elements where the elements are all the same type. The Backus-Naur form, or abstract syntax for a list is:

\[
(\alpha)\text{list} ::= [] | \alpha :: (\alpha)\text{list}
\]

The map2 function is a higher-order function which takes as its parameters a function \(f:\alpha \times \beta \rightarrow \gamma\), and two lists \(11:(\alpha)\text{list}\) and \(12:(\beta)\text{list}\). map2 is defined as:

\[
\begin{align*}
\text{map2 } f \ [\ ] \ l &= [\ ] \\
\text{map2 } f \ (x :: xs) \ (y :: ys) &= f(x, y) :: (\text{map2 } f \ xs \ ys)
\end{align*}
\]

For example, \(\text{map2 } + [1,2,3] \ [4,5,6]\) reduces to \([5,7,9]\).

The particular correctness property we are interested in as it applies to DES relates to inverses of bit-vectors. Specifically,

\[
(\forall \text{BitList } l1\ l2). (\text{length } l1 = \text{length } l2) \supset (l1 = \text{map2 } \oplus (\text{map2 } \oplus l1\ l2)\ l2)
\]

where BitList is a predicate which is true whenever it is applied to a list of numbers where each element is a Bit.

\[
\begin{align*}
\text{BitList } [\ ] &= \text{true} \\
\text{BitList } (b :: bs) &= (\text{Bit } b) \land (\text{BitList } bs)
\end{align*}
\]

Notice that (14) has as a condition that \(\text{length } l1 = \text{length } l2\). Why?

The proof of (14) is done by induction on \(l1\) and \(l2\), and using the property that

\[
\forall l.\text{length } [\ ] = \text{length } l \supset (l = [\ ])
\]
3 The Basic DES Round

The basic operation which is repeated sixteen times in DES is the DES Round. It is shown in Figure 2.

$L(i), R(i), L(i+1)$, and $R(i+1)$ are the left and right halves of the 64-bit word being encrypted by compressed key $K(i)$. Each left and right half is 32-bits wide. Compressed key $K(i)$ is 48-bits long and is derived from the initial 56-bit key.

Function $f$ is called the mangler function in [1]. It is a series of permutations and substitutions. The mangler function is what gives DES its cryptographic strength along with the initial key size of 56-bits. The mangler function is described in detail in later sections.

DES was originally designed for implementation in hardware. It has a simple hardware description in higher-order logic.

$$DESROUND(A, B, K, X, Y) = (X = B) \land (Y = \text{map2} \oplus A \ f(B, K)) \quad (18)$$

4 Decrypting with a DES Round

Decrypting ciphertext created by DES is done using the same DES function. DES is its own inverse if properly used. The basic theory which underlies decryption is given below.

\[
\begin{align*}
A0. & \quad x = x \\
A1. & \quad DESROUND(A, B, K, X, Y) = (X = B) \land (Y = map2 \oplus A \ f(B, K)) \\
A2. & \quad (\forall \text{BitList l1 l2})(\text{length l1} = \text{length l2}) \supset (l1 = map2 \oplus (map2 \oplus l1 \ l2)l2)
\end{align*}
\]

The property we can prove about DES decryption is:

$$\forall \text{BitList} \ \ L_{i+1}, R_{i+1}, K, L_i, R_i.$$  

$$(\text{BitList} \ f(R_i, K)) \supset (\text{length} \ L_i = \text{length} \ f(R_i, K)) \supset$$

$$DESROUND(L_i, R_i, K, L_{i+1}, R_{i+1}) \supset DESROUND(R_{i+1}, L_{i+1}, K, R_i, L_i)$$

What (19) says is if everything is a bit-vector of correct length, then the original 32-bit words $L_i$ and $R_i$ encrypted by DES using $K$ are obtained by taking the ciphertext $L_{i+1}$ and $R_{i+1}$ and inputting them into the right and left input ports $R_i$ and $L_i$, respectively. (19) gives a precise meaning to the phrase, “to invert DES just run it backwards,” which appears in some descriptions of DES.
5 Key Permutation

Each DES round uses a permutation of the previous round’s key.

5.1 Properties of Rotation

Rotation is the basic operation for permuting keys between each DES round. Rotation, while intuitively simple, is somewhat complex to reason about. The recursive definition, where “@” denotes the append operation on two lists, e.g., \([1;2]@[3;4] \rightarrow [1;2;3;4]\), is:

\[
\begin{align*}
\text{rotateLeft } 0 l & = l \\
\text{rotateLeft}(n+1)(x :: xs) & = \text{rotateLeft } n (xs@[x])
\end{align*}
\]

The above definition is an inconvenient way to reason about rotation when inducting over lists. (To see this, try using the above definition and attempt to prove the properties of rotate that follow).

Another way to define rotate in terms of take and drop where take n l takes the first n elements of a list while drop n l drops the first n elements.

Ultimately, we will show that:

- \(\text{rotateL } (\text{length } l) \ l = l\)
- \(\forall m \ n \ l. n + m \leq \text{length } l \supset \text{rotateL } n(\text{rotateL } m \ l) = \text{rotateL}(n + m)l\)
- \(\forall n \ l. (n \leq \text{length } l) \supset (\text{rotateRn}(\text{rotateL } n \ l) = l)\)

These properties allow us to prove:

- The key at the end of the final DES round is the same as the first.
- To “invert” the rotation of a DES round, one does a corresponding right rotation, or a \(\text{rotateL}((\text{length } l) - n)l\), to retrieve the key at the beginning of the round.

First, we define take.
Based on the definition of take we can prove:
\[
\forall l. \text{take}(\text{length } l)l = l
\] (22)

Next, we define drop, the complementary operation to take, as follows.

| A0. \( x = x \) |
| A1. \( \text{drop } 0 \ l = l \) |
| A2. \( \text{drop } n + 1 \ x :: xs = \text{drop } n \ xs \) |
| A3. \( \text{length } [] \) |
| A4. \( \text{length } x :: xs = 1 + \text{length } xs \) |

Given the definition of drop we can prove:
\[
\forall l. \text{drop}(\text{length } l) = []
\] (23)

Using the definitions of take and drop, we define \( \text{rotateL} \).

| A0. \( x = x \) |
| A1. \( \text{take } 0 \ l = [] \) |
| A2. \( \text{take } n + 1 \ x :: xs = x :: (\text{take } n \ xs) \) |
| A3. \( \text{take}(\text{length } l)l = l \) |
| A4. \( \text{drop } 0 \ l = l \) |
| A5. \( \text{drop } n + 1 \ x :: xs = \text{drop } n \ xs \) |
| A6. \( \text{drop}(\text{length } l) = [] \) |
| A7. \( \text{rotateL } n \ l = (\text{drop } n \ l)@(\text{take } n \ l) \) |

The next major property of rotation we want to prove is:
\[
\forall n \ x. x. n \leq \text{length } xs \ni \text{rotateL}(n + 1)(x :: xs) = \text{rotateL } n(xs@x[)] \] (24)

This corresponds to the normal notion of what rotation does, namely circulate bits from the head of the list to the tail of the list. To prove this property, we need the following:

\[
\forall l. x. x. l@x :: xs = (l@x)@xs
\] (25)

\[
\forall n. n \leq \text{length } l \ni \text{take } n \ (l@a[]) = (\text{take } n \ l)
\] (26)

\[
\forall n. x. x. n \leq \text{length } xs \ni (\text{drop } n \ xs)@x[] = \text{drop } n \ (xs@x[)]
\] (27)

(25) and (27) are relatively easy to prove. The proof of (26) is done by induction on \( n \).
Using the above properties, we can show: \( \forall n \ x \ s . n \leq \text{length } xs \supset \text{rotateL}(n + 1)(x :: xs) = \text{rotateL}(n(xs @ [x])) \).

It is a simple matter to prove:

\[
\forall l. \text{rotateL} \ 0 \ l = l
\]  

(28)

Based on the above properties, we can prove that the serial application of rotations is the same as applying the sum of all the rotations.

\[
\forall m \ n \ l. n + m \leq \text{length } l \supset \text{rotateL}(n(\text{rotateL} m \ l)) = \text{rotateL}(n + m)l
\]  

(29)

5.2 Retrieving the Key

Right rotation can be defined as the complement of left rotation. Doing so simplifies the proof of correctness of doing a right rotation to get back the key at the beginning of a DES round.

\[
\text{rotateR} n \ l = \text{rotateL}((\text{length } l) - n)l
\]  

(30)

Using the above definition, we can show:

\[
\forall n \ l. (n \leq \text{length } l) \supset (\text{rotateR} n(\text{rotateL} n \ l) = l)
\]  

(31)

Given the above property, to retrieve key\((i)\) from key\((i + 1)\) where \(n\) was the number of bits rotated left in DES round \(i\), we just take key\((i + 1)\) and do a right rotation of \(n\)-bits on each half to obtain key\((i)\) and the compressed 48-bit key \(K(i)\) used in the mangler function.

6 SML Implementation

The remaining sections show how the basic DES round is implemented in Standard ML of New Jersey, 0.93. SML is a widely used functional programming language. Note, the notation, \( \textbf{fn} \ x \Rightarrow x - 1 \) denotes a function \( \lambda x . x - 1 \), i.e. the decrement function. The \( \lambda x \) indicates that \( x \) is a bound variable or parameter in the expression \( x - 1 \) which follows. So, \( \lambda x . x - 1 \) is a function with parameter \( x \) that subtracts 1 from \( x \) when applied to an argument. For example, \( (\lambda x . x - 1)2 \rightarrow 2 - 1 \rightarrow 1 \). This type of function definition allows functions to be defined without having to name them. This is convenient for functions which are locally defined within a larger definition.

6.1 Basic Definitions

```sml
fun map2 f [] l = []
    | map2 f (x::xs) (y::ys) = f(x,y)::(map2 f xs ys);
```
exception PermListShort;

local fun perm ([: int list) (xs: 'a list) = ([: 'a list)
| perm (i :: is) xs = nth(xs,i) :: (perm is xs)
in fun permute ilist xlist =
    let val ilist’ = map (fn x => x - 1) ilist
    val ilen = List.length xlist
    in if exists (fn i => i > ilen) ilist
        then raise PermListShort
        else perm ilist’ xlist
    end
end;

exception rotL_num_neg and rotL_list Too small;

fun rotL num list =
    if num < 0
        then raise rotL_num_neg
        else if num > List.length list
            then raise rotL_list Too small
            else let val (x,y) = ListUtil.split num list
                   in y@x
                   end;

6.2 Initial and Final Permutation

val IP = [58, 50, 42, 34, 26, 18, 10, 2,
  60, 52, 44, 36, 28, 20, 12, 4,
  62, 54, 46, 38, 30, 22, 14, 6,
  64, 56, 48, 40, 32, 24, 16, 8,
  57, 49, 41, 33, 25, 17, 9, 1,
  59, 51, 43, 35, 27, 19, 11, 3,
  61, 53, 45, 37, 29, 21, 13, 5,
  63, 55, 47, 39, 31, 23, 15, 7];

val FP = [40, 8, 48, 16, 56, 24, 64, 32,
  39, 7, 47, 15, 55, 23, 63, 31,
  38, 6, 46, 14, 54, 22, 62, 30,
  37, 5, 45, 13, 53, 21, 61, 29,
  36, 4, 44, 12, 52, 20, 60, 28,
  35, 3, 43, 11, 51, 19, 59, 27,
  34, 2, 42, 10, 50, 18, 58, 26,
  33, 1, 41, 9, 49, 17, 57, 25];
6.3 Mangler Function Components

6.3.1 Expansion Permutation

``` scala
(* ----- Expansion Permutation ----- *)
val ExpPermutation =
[32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9,
  8, 9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17,
  16, 17, 18, 19, 20, 21, 20, 21, 22, 23, 24, 25,
  24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1];
```

6.3.2 S-Box Substitutions

``` scala
fun intToBinList numBits input =
  map (fn x => if x = "1" then 1 else 0) (intToBitString numBits input);

val S1 =
[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
  0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
  4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
  15, 12, 8, 2, 4, 9, 1, 5, 11, 3, 14, 10, 0, 6, 13];

val S1BitList =
  [[1,1,1],[0,1,0],[1,1,0],[0,0,0],[0,0,1],[1,1,1],[1,0,1],[1,0,0],[0,1,1],[0,1,0],
   [0,0,1],[0,1,1],[1,0,0],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,0,1],[0,0,0],
   [1,1,1],[1,1,0],[0,1,1],[0,1,0],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0]],

val S2 =
[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
  3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
  0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
  13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9];

val S2BitList =
  [[1,1,1],[0,0,1],[1,0,0],[0,1,0],[1,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,1],[0,1,0],
   [0,0,1],[1,1,0],[1,1,1],[0,1,0],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,1,1],[1,1,0],
   [0,0,1],[1,0,0],[0,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],
   [0,0,1],[0,0,0],[1,1,1],[1,1,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],
   [0,1,0],[0,1,1],[1,0,1],[0,1,0],[0,1,1],[1,0,1],[1,0,0],[0,1,0],[0,1,1],[1,0,1],[1,0,0]],
```

CSE 791: E-3

DES: 9
val S3 =
[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12];

val S3BitList =
[[[1,0,1,0], [0,0,0,0], [1,0,0,1], [1,1,1,0], [0,1,1,0], [0,0,1,1], [1,1,1,1],
[0,1,0,1], [0,0,0,1], [1,1,0,1], [1,1,0,0], [0,1,1,1], [1,0,1,1], [0,1,0,0],
[0,0,1,0], [1,0,0,0], [1,1,0,1], [0,1,1,0], [0,0,0,0], [1,0,0,1], [0,1,1,1],
[0,1,0,0], [0,1,1,0], [1,0,0,0], [1,1,0,0], [0,1,0,1], [1,1,1,0], [1,1,1,0],
[1,1,0,1], [1,1,1,0], [0,0,0,1], [1,1,0,1], [0,1,1,0], [0,1,0,0],
[0,0,0,0], [1,1,1,1], [0,0,1,1], [0,0,0,0], [1,1,1,0], [0,1,1,1], [0,0,1,1],
[0,1,1,1], [1,1,0,0], [1,0,1,1], [0,1,1,0], [0,0,1,0], [1,0,1,0], [1,0,0,1],
[0,0,0,1], [1,1,0,0], [0,1,0,0], [1,0,1,1], [0,1,0,1], [1,1,1,1], [0,0,0,1],
[0,0,1,1], [1,1,1,0], [0,1,0,0], [1,0,0,0], [1,1,0,0], [0,0,0,0], [0,0,0,0],
[1,1,0,1], [0,1,0,0], [0,1,0,1], [1,1,0,0], [1,0,0,0], [0,1,1,1], [0,0,1,0],
[1,0,0,1], [0,1,0,0], [0,0,1,1], [1,0,0,1], [1,1,0,0], [0,1,1,0], [0,1,1,0],
[1,1,0,0]] : int list list

val S4 =
[ 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14];

val S4BitList =
[[[0,1,1,1], [1,1,0,1], [1,1,1,0], [0,0,1,1], [0,0,0,0], [0,1,1,0], [1,0,0,1],
[1,0,1,0], [0,0,0,1], [0,0,1,0], [1,0,0,0], [0,1,0,1], [1,0,1,1], [0,1,1,0],
[0,1,0,0], [0,1,1,1], [1,1,0,1], [1,0,0,0], [1,0,1,1], [0,1,1,0], [1,0,1,0],
[1,1,1,1], [0,0,0,0], [0,0,1,1], [0,1,0,0], [0,1,1,0], [0,0,1,0], [1,1,0,0],
[0,0,0,1], [1,0,1,0], [1,1,1,0], [1,0,0,1], [1,0,1,0], [0,1,1,0], [1,0,0,1],
[1,0,0,1], [1,1,0,0], [0,1,0,1], [0,1,0,0], [1,0,0,1], [1,1,1,1], [0,0,0,1],
[0,0,1,1], [1,1,1,0], [0,1,0,0], [0,1,0,0], [0,0,0,0], [1,1,0,1], [1,1,0,1],
[1,1,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,1], [1,1,0,1], [1,1,0,0], [1,0,0,0],
[1,0,0,1], [0,1,0,0], [0,1,0,1], [1,0,1,1], [1,1,0,0], [0,1,1,0], [0,1,1,0],
[1,1,0,0]] : int list list
val S5 =
[ 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
 4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3];

val S5BitList =
[[0,0,1,0],[1,1,0,0],[0,1,0,0],[0,0,0,1],[0,1,1,1],[1,0,1,0],[1,0,1,1],
 [0,1,1,0],[1,0,0,0],[0,1,0,1],[0,0,1,1],[1,1,1,1],[1,1,0,1],[0,0,0,0],
 [1,1,1,0],[1,0,0,1],[1,1,1,0],[1,0,1,1],[0,0,1,0],[1,1,0,0],[0,1,0,0],
 [0,1,1,1],[1,1,0,1],[0,0,0,1],[0,1,0,1],[0,0,0,0],[1,1,1,1],[1,0,1,0],
 [0,0,1,1],[1,0,0,1],[1,0,0,0],[0,1,1,0],[0,1,0,0],[0,0,1,0],[0,0,0,1],
 [1,0,1,1],[1,0,1,0],[1,0,0,1],[1,1,1,1],[1,0,1,0],[1,0,0,0],[1,1,0,1],
 [1,1,0,0],[0,1,0,1],[0,0,1,1],[0,0,0,0],[1,1,0,0],[1,0,1,0],[0,1,0,0],[0,0,1,1],
 [0,1,1,0],[1,1,1,1],[0,0,0,0],[1,0,0,1],[1,0,1,0],[0,1,0,0],[0,1,0,1],
 [0,0,1,1],[1,1,1,1],[0,0,0,0],[1,0,0,1],[1,0,1,0],[0,1,0,0],[0,1,0,1],
 [0,0,1,1],[1,1,1,1],[0,0,0,0],[1,0,0,1],[1,0,1,0],[0,1,0,0],[0,1,0,1],
 [0,0,1,1],[1,1,1,1],[0,0,0,0],[1,0,0,1],[1,0,1,0],[0,1,0,0],[0,1,0,1]] : int list list

val S6 =
[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
 9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13];

val S6BitList =
[[1,1,0,0],[0,0,0,1],[1,0,1,0],[1,1,1,1],[1,0,0,1],[0,0,1,0],[0,1,1,0],
 [1,0,0,1],[0,0,0,0],[1,1,0,1],[0,0,1,1],[0,1,0,0],[1,1,0,0],[0,1,1,0],
 [0,1,0,1],[1,0,1,1],[1,0,0,1],[1,1,1,1],[1,0,1,0],[0,0,1,0],[0,1,1,1],
 [1,1,1,0],[1,0,0,1],[1,0,0,0],[0,1,0,1],[0,1,1,0],[0,0,0,1],[1,1,0,1],
 [0,0,0,0],[1,0,1,1],[0,0,1,1],[1,0,0,0],[1,0,0,1],[1,1,1,0],[1,1,1,1],
 [0,1,0,1],[0,0,0,1],[1,0,0,0],[1,1,0,0],[0,0,1,1],[0,1,1,1],[0,0,0,0],
 [0,1,0,1],[1,0,1,0],[0,0,0,1],[1,1,0,1],[0,0,1,1],[1,0,0,1],[1,1,0,0],
 [0,0,1,1],[0,0,0,0],[1,1,0,0],[0,0,0,1],[0,1,0,1],[1,1,1,1],[1,0,0,0],
 [1,0,1,1],[1,1,0,0],[0,0,0,1],[0,1,1,1],[0,1,1,0],[0,0,0,0],[1,0,0,0],
 [1,1,0,1]] : int list list
val S7 =
[ 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
 1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12];

val S7BitList =
[[0,1,0,0], [1,0,1,1], [0,0,1,0], [1,1,1,0], [0,1,1,1], [0,0,0,0], [1,0,0,0],
 [1,1,1,0], [0,0,1,1], [1,1,0,0], [1,0,0,1], [0,1,1,1], [0,1,0,1], [1,0,1,0],
 [0,0,1,0], [0,0,0,1], [1,1,0,1], [1,0,0,0], [1,0,1,1], [0,1,1,1], [0,1,0,0],
 [1,0,0,1], [0,0,1,0], [1,1,1,0], [0,0,0,1], [0,1,1,1], [1,1,0,0], [0,1,0,1], [1,1,1,1],
 [0,0,1,1], [1,1,1,0], [0,1,0,0], [0,1,1,1], [0,0,1,0], [0,1,1,0], [1,1,0,0], [0,1,1,1],
 [1,0,1,0], [0,1,0,1], [0,0,0,0], [1,1,1,1], [0,0,1,0], [0,1,1,0], [1,1,1,0], [0,0,0,1], [1,1,1,1],
 [0,0,1,0], [1,1,1,0], [0,0,0,0], [0,1,1,1], [0,0,1,0], [0,1,1,0], [1,1,1,0], [0,0,0,1], [1,1,1,0],
 [0,0,1,0], [1,1,1,0], [0,0,0,0], [0,1,1,0], [0,1,0,0], [1,1,0,0], [0,1,1,0], [0,1,0,1], [1,1,0,1],
 [1,1,1,0], [0,1,0,1], [0,0,0,0], [0,1,1,1], [0,0,1,1], [0,1,0,1], [0,1,0,1], [0,1,1,0], [0,1,1,0],
 [1,1,1,1], [1,1,0,0], [1,0,0,1], [0,0,0,0], [0,0,1,1], [0,1,0,1], [0,1,1,0], [0,1,1,0],
 [1,0,1,1]] : int list list

val S8 =
[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
 7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
 2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11];

val S8BitList =
[[1,1,0,1], [0,0,1,0], [1,0,0,0], [0,1,0,0], [0,1,1,0], [1,1,1,1], [1,0,1,1],
 [0,0,0,1], [1,0,1,0], [1,1,0,1], [0,0,1,1], [1,1,1,0], [1,0,1,0], [0,0,0,0],
 [1,1,1,0], [0,1,0,1], [0,0,0,1], [1,1,1,1], [1,1,0,1], [1,0,0,0], [1,0,1,0],
 [0,0,0,1], [0,1,1,1], [0,1,0,0], [1,1,0,0], [0,1,0,1], [0,1,1,0], [1,0,0,1],
 [0,0,0,0], [1,1,1,0], [1,0,0,1], [0,0,1,0], [0,1,1,1], [1,0,1,1], [0,1,0,0],
 [0,0,0,1], [1,0,0,1], [1,1,1,0], [0,0,1,0], [1,1,1,0], [0,0,0,0], [0,1,0,0], [0,1,1,0],
 [1,0,1,0], [1,1,1,1], [0,0,1,1], [0,1,0,0], [1,1,0,1], [1,0,0,1], [0,0,0,0], [0,1,0,0],
 [0,0,0,1], [1,1,1,0], [0,1,1,1], [0,1,0,0], [1,0,1,0], [1,1,0,0], [1,1,0,1],
 [1,1,1,1], [1,1,0,0], [1,0,0,1], [0,0,0,0], [0,0,1,1], [0,1,0,1], [0,1,0,1],
 [1,0,1,0], [1,0,1,1]] : int list list
6.3.3 P-Box Permutation

```plaintext
(* ----- P-Box Permutation ----- *)
val PBoxPermutation =
[16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10,
  2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25];
```

6.4 Key Permutation

```plaintext
(* ----- Number of Key Bits Shifted per Round ----- *)
val Shift = [1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1];
```

Each *eighth* bit of the 64-bit key is used for *parity* and technically is not part of the key. So, the first permutation not only permutes the order of key bits, but omits bits 8, 16, 24, 32, 40, 48, 56, and 64. The initial key permutation is given below.
The compression permutation is used each round to generate a 48-bit key from the rotated 56-bit key.

7 DES Round Functions

7.1 Key Rotation

(* The Mangler Function consists of a 32-bit to 48-bit expansion *)
(* of R using the expansion permutation ExpPermutation exor’d *)
(* with a compression permutation of the key with CompPermutation. *)
(* We do the permutations in a bit-wise fashion and then convert *)
(* the result to 8 integer values obtained 6-bits at a time. The *)
(* integer values access the 8 S-boxes (origin 0). *)

(* RotateComp n key splits, rotates, and compresses a 56-bit key *)
(* to produce a 8 integer values corresponding to the 8 6-bit *)
(* values exor’d with the data and fed into the S-boxes. n is the *)
(* number of left rotations and key is the 56-bit key. *)
(* The key is a list of 56 0’s and 1’s. *)

except BadRotationValue and KeyMustBe56Bits;
fun RotateComp n key =
  if not (n = 1 orelse n = 2) then raise BadRotationValue
  else if List.length key <> 56 then raise KeyMustBe56Bits
     else
        let val (C,D) = ListUtil.split 28 key
        val newKey = (rotL n C)@ (rotL n D)
        val compKey = permute CompPermutation newKey
        in (newKey,compKey)
   end;
7.2 Mangler Function

(* Mangler function *)
fun mangle R K =
  let val 11 = permute ExpPermutation R
  val 12 = map2 Bits.xorb 11 K
  val 13 = SBoxSubstitution 12
  in permute PBoxPermutation 13
end;

7.3 DES Round

fun DESRound n (L,R,K) =
  let val (newKey,compKey) = RotateCompKey n K
       val 11 = mangle R compKey
       val 12 = map2 Bits.xorb L 11
  in (R,12,newKey)
end;

8 Problems

The following problems are proofs of properties used in the previous sections. When attempting a proof of a particular problem, it is permissible to use the properties of the previous problems, i.e., it is all right to assume the properties of the previous problems have been proved.

1. Prove: (\( \forall \ Bit \ x \). x \( \oplus \) x = 0

2. Prove: (\( \forall \ Bit x \). x \( \oplus \) 0 = x

3. Prove: (\( \forall \ Bit x \ y \). x \( \oplus \) y = y \( \oplus \) x

4. Prove: (\( \forall \ Bit x \ y \ z \). (x \( \oplus \) y) \( \oplus \) z = x \( \oplus \) (y \( \oplus \) z)

5. Prove: (\( \forall \ Bit x \ y \). [x = (x \( \oplus \) y) \( \oplus \) y]

6. Prove: \( \forall l. length [ ] = length l \supset (l = [ ])

7. Prove: (\( \forall \ BitList \ l1 \ l2 \). (length l1 = length l2) \supset (l1 = map2 \( \oplus \) (map2 \( \oplus \) l1 l2) l2)

8. Prove:
   (\( \forall \ BitList \ L_{i+1} \ R_{i+1} \ K \ L_i \ R_i \).
    (BitList f(R_i, K)) \supset (length L_i = length f(R_i, K)) \supset
    DESROUND(L_i, R_i, K, L_{i+1}, R_{i+1}) \supset DESROUND(R_{i+1}, L_{i+1}, K, R_i, L_i)

9. Prove: \( \forall l. take(length l) l = l

10. Prove: \( \forall l. drop(length l) = [ ].

CSE 791: E-3 DES: 15
11. Prove: \( \forall l. \text{rotate}(l, l) = l \)
12. Prove: \( \forall l \, x \, x \cdot x \cdot \text{rotate}(l, x) = (l \cdot x) \cdot x \)
13. Prove: \( \forall n \, l. (n \leq \text{length } l) \supset \text{take}(n, (l \cdot [a])) = (\text{take } n \, l) \)
14. Prove: \( \forall n \, x \, x \cdot n \leq \text{length } x \cdot (\text{drop } n \, x \cdot [x] = \text{drop } n \, (x \cdot [x])) \)
15. Prove: \( \forall n \, x \, x \cdot n \leq \text{length } x \cdot (\text{rotate}(n + 1, x) = \text{rotate}(n, x \cdot [x])) \)
16. Prove: \( \forall l. \text{rotate}(l, 0, l) = l \)
17. Prove: \( \forall m \, n \, l. n + m \leq \text{length } l \supset \text{rotate}(n, \text{rotate}(m, l)) = \text{rotate}(n + m, l) \)
18. Prove: \( \forall n \, l. (n \leq \text{length } l) \supset (\text{rotate}R \, n \cdot \text{rotate}(n, l) = l) \)

9 Solutions

Problem 5

A0. \( x = x \)
A1. (\( \forall \text{Bit } x \cdot x \) \( \oslash \) \( x = 0 \)
A2. (\( \forall \text{Bit } x \cdot x \) \( \oslash \) 0 = \( x \)
A3. (\( \forall \text{Bit } x \cdot y \cdot x \) \( \oslash \) \( y = y \) \( \oslash \) \( x \)
A4. (\( \forall \text{Bit } x \cdot y \cdot z \cdot (x \) \( \oslash \) \( y \) \( \oslash \) \( z = x \) \( \oslash \) \( (y \) \( \oslash \) \( z \)
G1. (\( \forall \text{Bit } x \cdot y \cdot x = (x \) \( \oslash \) \( y \) \( \oslash \) \( y \)
G2. \( \forall x \cdot y \cdot (\text{Bit } x) \supset (\text{Bit } y) \supset x = (x \) \( \oslash \) \( y \) \( \oslash \) \( y \)
G3. (Bit \( x' \) \( \supset \) Bit \( y' \) \( \supset \) \( x' = (x' \) \( \oslash \) \( y' \) \( \oslash \) \( y' \)
A5. Bit \( x' \)
A6. Bit \( y' \)
G4. \( x' = (x' \) \( \oslash \) \( y' \) \( \oslash \) \( y' \)
A7. Bit \( x \supset \) Bit \( y \supset \) Bit \( z \supset (x \) \( \oslash \) \( y \) \( \oslash \) \( z = x \) \( \oslash \) \( (y \) \( \oslash \) \( z \)
A8. (x' \( \oslash \) y') \( \oslash \) z' = x' \( \oslash \) (y' \( \oslash \) z')
A9. Bit \( x \supset x \) \( \oslash \) x = 0
A10. y' \( \oslash \) y' = 0
A11. (x' \( \oslash \) y') \( \oslash \) y' = x' \( \oslash \) 0
A12. Bit \( x \supset x \) \( \oslash \) 0 = x
A13. x' \( \oslash \) 0 = x'
A14. (x' \( \oslash \) y') \( \oslash \) y' = x'
G5. x' \( = x' \)
G6. true
Problem 7

A0. \( x = x \)
A1. \((\forall \ Bit \ x \ y). x = (x \oplus y) \oplus y\)
A2. \(\map f \ [ ] = [ ]\)
A3. \(\map f \ (x :: x \ 8) \ = f \ (x, y) :: (map2 \ f \ x \ 8 \ y)\)
A4. \(\text{BitList} [ ] = \text{true}\)
A5. \(\text{BitList} (b :: bs) = (\text{Bit} \ b) \land (\text{BitList} bs)\)

G1. \((\forall \text{BitList} \ l1 \ l2). (\text{length} \ l1 = \text{length} \ l2) \lor (l1 = \map2 \oplus (\map2 \oplus \text{length} \ l2) / 2)\)

The base case.

Base case

G2. \((\forall l1 \ l2). \text{BitList} \ l1 \supset \text{BitList} \ l2 \supset (\text{length} \ l1 = \text{length} \ l2) \lor (l1 = \map2 \oplus (\map2 \oplus \text{length} \ l2) / 2)\)
G3. \((\forall l1 \ l2). \text{BitList} \ l1 \supset \text{BitList} \ l2 \supset (\text{length} \ l1 = \text{length} \ l2) \lor (l1 = \map2 \oplus (\map2 \oplus \text{length} \ l2) / 2)\)
G4. \((\forall l1 \ l2). \text{BitList} \ l1 \supset \text{BitList} \ l2 \supset (\text{length} \ l1 = \text{length} \ l2) \lor (l1 = \map2 \oplus (\map2 \oplus \text{length} \ l2) / 2)\)
A6. \(\text{BitList} [ ]\)
A7. \(\text{BitList} l2\)
A8. \(\text{length} [ ] = \text{length} \ l2\)

G5. \(\text{length} [ ] = \map2 \oplus (\map2 \oplus [ ] / 2)\)
G6. \(\text{length} [ ] = [ ]\)
G7. \(\text{true}\)

The induction-step.

G3.2 \[\forall l1. (\text{BitList} l1' \supset \text{BitList} l2 \supset (\text{length} l1' = \text{length} l2) \lor \left[ l1' = \map2 \oplus (\map2 \oplus l1' / 2) \right] \)

G4.2 \[\forall l1. (\text{BitList} a :: l1' \supset \text{BitList} l2 \supset (\text{length} a :: l1' = \text{length} l2) \lor \left[ a :: l1' = \map2 \oplus (\map2 \oplus a :: l1' / 2) \right] \)

A6.2 \(\text{BitList} l1' \supset \text{BitList} l2 \supset (\text{length} l1' = \text{length} l2) \lor l1' = \map2 \oplus (\map2 \oplus l1' / 2)\)
A6.2 \(\text{BitList} l1' \supset \text{BitList} l2 \supset (\text{length} a :: l1' = \text{length} l2) \lor a :: l1' = \map2 \oplus (\map2 \oplus a :: l1' / 2)\)

We solve the induction step by induction on \( l2 \) and by using the property that:

\(\text{length} \ a :: l1' \neq \text{length} [ ]\)

The base case of \( l2 \).

G5.2.1 \[\text{BitList} a :: l1' \supset \text{BitList} [ ] \supset (\text{length} a :: l1' = \text{length} [ ] \lor \left[ a :: l1' = \map2 \oplus (\map2 \oplus a :: l1' / 2) \right] \)

G6.2.1 \(\text{true}\)
The induction step of \( l_2 \).

\[
G5.2.2 \quad (\text{BitList } a \equiv l_1 \triangleright \text{BitList } l_2 \triangleright (\text{length } a \equiv l_1' = \text{length } l_2') \triangleright
a \equiv l_1' = \text{map2} \oplus (\text{map2} \oplus a \equiv l_1' \ l_2')
\]

\[
G7.2.2 \quad a \equiv l_1' = \text{map2} \oplus (\text{map2} \oplus a \equiv l_1' \ b \equiv l_2')
\]

\[
G8.2.2 \quad a \equiv l_1' = \text{map2} \oplus ((a \oplus b) \equiv (\text{map2} \oplus l_1' \ l_2'))
\]

\[
G9.2.2 \quad a \equiv l_1' = ((a \oplus b) \oplus b) \equiv \text{map2} \oplus (\text{map2} \oplus l_1' \ l_2')
\]

\[
A11.2.2 (\text{Bit } a) \land \text{BitList } l_1'
\]

\[
A13.2.2 (\text{Bit } b) \land \text{BitList } l_2'
\]

\[
A14.2.2 (\text{length } l_1') = (\text{length } l_2')
\]

\[
A15.2.2 (\text{Bit } x) \equiv (\text{Bit } y) \equiv (x \oplus y) \equiv y
\]

\[
A16.2.2 a = (a \oplus b) \oplus b
\]

\[
G10.2.2 a \equiv l_1' = a \equiv \text{map2} \oplus (\text{map2} \oplus l_1' \ l_2')
\]

\[
A17.2.2 l_1' = \text{map2} \oplus (\text{map2} \oplus l_1' \ l_2') \text{ from } A6.2
\]

\[
G11.2.2 a \equiv l_1' = a \equiv l_1'
\]

\[
G12.2.2 \text{ true}
\]

**Problem 8**

First, we set up the goal and do the normal simplifications involving relativized quantifiers, quantifier removal, and if-splitting.
A0. \( x = x \)

A1. \( \text{DESRound}(A, B, K, X, Y) = (X = B) \land (Y = \text{map2} \oplus A \ f(B, K)) \)

A2. \( (\forall \text{BitList} \ l \ l_2, (\text{length} \ l_1 = \text{length} \ l_2) \supset (l_1 = \text{map2} \oplus (\text{map2} \oplus l_1 \ l_2)l_2) \)

\[
\begin{align*}
G1. & \quad (\forall \text{BitList} \ L_{i+1} K \ L_i R_i) \\
& \quad (\text{BitList} \ f(R_i, K)) \supset (\text{length} \ L_i = \text{length} \ f(R_i, K)) \supset \text{DESRound}(L_i, R_i, K, L_{i+1}, R_{i+1}) \supset \text{DESRound}(\text{R}_{i+1}, L_{i+1}, K, R_i, L_i) \\
G2. & \quad (\forall \text{BitList} \ L_{i+1} \supset \text{BitList} \ R_{i+1} \supset \text{BitList} \ K \supset \text{BitList} \ L_i \supset \text{BitList} \ R_i \supset \text{DESRound}(L_i, R_i, K, L_{i+1}, R_{i+1}) \supset \text{DESRound}(\text{R}_{i+1}, L_{i+1}, K, R_i, L_i) \\
G3. & \quad (\forall \text{BitList} \ L_{i+1} \supset \text{BitList} \ R_{i+1} \supset \text{BitList} \ K' \supset \text{BitList} \ L_i \supset \text{BitList} \ R_i \supset \text{DESRound}(L_i, R_i, K', L_{i+1}, R_{i+1}) \supset \text{DESRound}(\text{R}_{i+1}, L_{i+1}, K', R_i, L_i) \\
\end{align*}
\]

A3. \( \text{BitList} \ L'_{i+1} \)

A4. \( \text{BitList} \ R'_{i+1} \)

A5. \( \text{BitList} \ K' \)

A6. \( \text{BitList} \ L_i \)

A7. \( \text{BitList} \ R_i \)

A8. \( \text{BitList} \ f(R'_i, K') \)

A9. \( \text{length} \ L'_i = \text{length} \ f(R'_i, K') \)

A10. \( \text{DESRound}(L'_i, R'_i, K', L'_{i+1}, R'_{i+1}) \supset \text{DESRound}(\text{R'}_{i+1}, L'_{i+1}, K', R'_i, L'_i) \)

The remainder of the proof is done by rewriting using the definition of DESRound and the inversion property of \( \oplus \) on bit-vectors of equal length.

\[
\begin{align*}
A11. & \quad L'_{i+1} = R'_i \\
A12. & \quad R'_{i+1} = \text{map2} \oplus L'_i \ f(R'_i, K') \\
& \quad G5. (R'_i = L'_{i+1}) \land (L'_i = \text{map2} \oplus R'_{i+1} \ f(L'_{i+1}, K')) \\
& \quad G6. L'_i = \text{map2} \oplus R'_{i+1} \ f(L'_{i+1}, K') \\
& \quad G7. L'_i = \text{map2} \oplus (\text{map2} \oplus L'_i \ f(R'_i, K')) \ f(R'_i, K') \\
A13. & \quad \text{BitList} \ l \supset \text{BitList} \ l_2 \supset \text{length} \ l_1 = \text{length} \ l_2 \supset (l_1 = \text{map2} \oplus (\text{map2} \oplus l_1 \ l_2)l_2) \\
A14. & \quad l' = \text{map2} \oplus (\text{map2} \oplus l' \ l'')l'' \\
& \quad G8. L'_i = L'_i \\
& \quad G9. \text{true}
\end{align*}
\]
Problem 9

A0. \( x = x \)
A1. take 0 \( l = [] \)
A2. take \( n + 1 \ x :: xs = x :: (\text{take} \ n \ xs) \)
A3. length \( [] \)
A4. length \( x :: xs = 1 + \text{length} \ xs \)
  \( G1. \ \forall l. \text{take}(\text{length} \ l) l = l \)
Base Case
  \( G2.1 \ \text{take}(\text{length} \ []) [] = [] \)
  \( G3.1 \ \text{take} 0 [] = [] \)
  \( G4.1 [] = [] \)
  \( G5.1 \ \text{true} \)
Induction Step
  \( G2.2 \ \text{take}(\text{length} \ l') l' :\text{take} (\text{length} \ a :: l') a :: l' = a :: l' \)
A5.2 take(\text{length} \ l') l' = l'
  \( G3.2 \ \text{take}(\text{length} \ a :: l') a :: l' = a :: l' \)
  \( G4.2 \ \text{take}(\text{length} \ l') + 1 a :: l' = a :: l' \)
  \( G5.2 \ a :: (\text{take}(\text{length} \ l') l') = a :: l' \)
  \( G6.2 \ a :: l' = a :: l' \)
  \( G7.2 \ \text{true} \)

Problem 10

A0. \( x = x \)
A1. drop 0 \( l = l \)
A2. drop \( n + 1 \ x :: xs = \text{drop} \ n \ xs \)
A3. length \( [] \)
A4. length \( x :: xs = 1 + \text{length} \ xs \)
  \( G1. \ \forall l. \text{drop}(\text{length} \ l) l = [] \)
Base Case
  \( G2.1 \ \text{drop}(\text{length} \ []) [] = [] \)
  \( G3.1 \ \text{drop} 0 [] = [] \)
  \( G4.1 [] = [] \)
  \( G5.1 \ \text{true} \)
Induction Step
  \( G2.2 \ \text{drop}(\text{length} \ l') l' = [] \cup \text{drop}(\text{length} \ a :: l') a :: l' = [] \)
A5.2 drop(\text{length} \ l') l' = []
  \( G3.2 \ \text{drop}(\text{length} \ a :: l') a :: l' = [] \)
  \( G4.2 \ \text{drop}(\text{length} \ l') + 1 a :: l' = [] \)
  \( G5.2 \ \text{drop}(\text{length} \ l') l' = [] \)
  \( G6.2 \ \text{true} \)
Problem 11

A0. $x = x$
A1. take $0 \cdot l = []$
A2. take $n + 1 \cdot x :: xs = x :: (\text{take } n \cdot xs)$
A3. take$(\text{length } l)\cdot l = l$
A4. drop $0 \cdot l = l$
A5. drop $n + 1 \cdot x :: xs = \text{drop } n \cdot xs$
A6. drop$(\text{length } l) = []$
A7. rotateL \cdot n \cdot l = (\text{drop } n \cdot l)@((\text{take } n \cdot l)$
G1. $\forall l. \text{rotateL}(\text{length } l)\cdot l = l$
G2. rotateL$(\text{length } l')\cdot l' = l'$
G3. (drop$(\text{length } l')\cdot l')@((\text{take } (\text{length } l')\cdot l') = l'$
G4. $[]@l' = l'$
G5. $l' = l'$
G6. true

Problem 13

A0. $x = x$
A1. take $0 \cdot l = []$
A2. take $n + 1 \cdot x :: xs = x :: (\text{take } n \cdot xs)$
G1. $\forall l. (n \leq \text{length } l) \supset \text{take } n \cdot (l@[a]) = (\text{take } n \cdot l)$

**Base Case**
G2.1 $\forall l. (0 \leq \text{length } l) \supset \text{take } 0 \cdot (l@[a]) = (\text{take } 0 \cdot l)$
G3.1 $[] = []$
G4.1 true

**Induction Step**
G2.2 $[\forall l. n' \leq \text{length } l \supset \text{take } n' \cdot (l@[a]) = \text{take } n' \cdot l]$
G3.2 $\forall l. n' + 1 \leq \text{length } l \supset \text{take } n' + 1 \cdot (l@[a]) = \text{take } n' + 1 \cdot l$

**Base Case**
G4.2.1 $n' + 1 \leq \text{length } [] \supset \text{take } n' + 1 \cdot ([]@[a]) = \text{take } n' + 1 \cdot []$
G5.2.1 true$(n' + 1 \leq 0)$

**Induction step**
G4.2.2 $[n' + 1 \leq \text{length } l' \supset \text{take } n' + 1 \cdot (l'@[a]) = \text{take } n' + 1 \cdot l']$
G5.2.2 $n' + 1 \leq \text{length } b :: l' \supset \text{take } n' + 1 \cdot (b :: l'@[a]) = \text{take } n' + 1 \cdot b :: l'$
G6.2.2 $\text{take } n' + 1 \cdot (b :: l'@[a]) = \text{take } n' + 1 \cdot b :: l'$
G7.2.2 $b :: (\text{take } n' \cdot (l'@[a]) = b :: (\text{take } n' \cdot l')$
G8.2.2 $b :: (\text{take } n' \cdot l') = b :: (\text{take } n' \cdot l')$
G9.2.2 true
Problem 15

A0. \( x = x \)
A1. \( \text{take } 0 \ l = [\] \)
A2. \( \text{take } n + 1 \ x :: xs = x :: (\text{take } n \ xs) \)
A3. \( \text{take(length } \ l) \ l = \ l \)
A4. \( \text{drop } 0 \ l = \ l \)
A5. \( \text{drop } n + 1 \ x :: xs = \text{drop } n \ xs \)
A6. \( \text{drop(length } \ l) \ l = [\] \)
A7. \( \text{rotateL} \ n \ l = (\text{drop } n \ l)@(\text{take } n \ l) \)
A8. \( \text{rotateL(length } \ l) \ l = \ l \)
A9. \( l@(x :: xs) = (l@[x])@xs \)
A10. \( (n \leq \text{length } \ l) \supset \text{take } n \ (l'[@[a]]) = (\text{take } n \ l) \)
A11. \( n \leq \text{length } \ xs \supset (\text{drop } n \ xs)@[x] = \text{drop } n \ (xs@[x]) \)
    G1. \( \forall n \ x \ xs.n \leq \text{length } \ xs \supset \text{rotateL}(n + 1)(x :: xs) = \text{rotateL} n(xs@[x]) \)
    G2. \( n' \leq \text{length } \ xs' \supset \text{rotateL}(n' + 1)(x' :: xs') = \text{rotateL} n'(xs'@[x']) \)
A12. \( n' \leq \text{length } \ xs' \)
    G3. \( \text{rotateL}(n' + 1)(x' :: xs') = \text{rotateL} n'(xs'@[x']) \)
    G4. \( (\text{drop}(n' + 1)(x' :: xs'))@(\text{take}(n' + 1)(x' :: xs')) = (\text{drop} n'(xs'@[x']))@(\text{take} n'(xs'@[x'])) \)
    G5. \( (\text{drop} n' \ xs')@[x' :: (\text{take} n' \ xs')] = (\text{drop} n'(xs'@[x']))@(\text{take} n'(xs'@[x'])) \)
    G6. \( (\text{drop} n' \ xs')@[x']@(\text{take} n' \ xs') = (\text{drop} n'(xs'@[x']))@(\text{take} n'(xs'@[x'])) \)
A13. \( (\text{drop} n \ xs')@[x'] = \text{drop} n \ (xs'@[x']) \)
    G7. \( (\text{drop} n \ (xs'@[x']))@(\text{take} n' \ xs') = (\text{drop} n'(xs'@[x']))@(\text{take} n'(xs'@[x'])) \)
    G8. true
Problem 17

A0. $x = x$
A1. $n \leq \text{length } xs \supset \text{rotate } L(n + 1)(x :: xs) = \text{rotate } L n(xs @ [x])$
A2. $\text{rotate } L 0 l = l$
   G1. $\forall m \ n \ . \ n + m \leq \text{length } l \supset \text{rotate } L n(\text{rotate } L m l) = \text{rotate } L(n + m)l$
Base Case
   G2.1 $\forall n \ . \ n \leq \text{length } l \supset \text{rotate } L 0(\text{rotate } L n l) = \text{rotate } L n l$
   G3.1 $n' \leq \text{length } l' \supset \text{rotate } L 0(\text{rotate } L n' l') = \text{rotate } L n' l'$
   A.3.1 $n' \leq \text{length } l'$
   G4.1 $\text{rotate } L 0(\text{rotate } L n' l') = \text{rotate } L n' l'$
   G5.1 $\text{rotate } L n' l' = \text{rotate } L n' l'$
   G6.1 true
Induction Step
   G2.2 $[\forall n \ . \ n + m' \leq \text{length } l \supset \text{rotate } L n(\text{rotate } L m' l) = \text{rotate } L(n + m')l]$
   $\supset$
   $[\forall n \ . \ n + m' + 1 \leq \text{length } l \supset \text{rotate } L n(\text{rotate } L m' + 1 l) = \text{rotate } L(n + m' + 1)l]$
A3.2 $\forall n \ . \ n + m' \leq \text{length } l \supset \text{rotate } L n(\text{rotate } L m' l) = \text{rotate } L(n + m')l$
   G3.2 $\forall n \ . \ n + m' + 1 \leq \text{length } l \supset \text{rotate } L n(\text{rotate } L m' + 1 l) = \text{rotate } L(n + m' + 1)l$
   G4.2 $\forall n \ . \ n + m' + 1 \leq \text{length } l \supset \text{rotate } L n'(\text{rotate } L m' + 1 l) = \text{rotate } L(n' + m' + 1)l$
Base Case
   G5.2.1 $n' + m' + 1 \leq \text{length } [[]\supset \text{rotate } L n'(\text{rotate } L m' + 1 [[]) = \text{rotate } L(n' + m' + 1)[[]]$
   G6.2.1 true $(n' + m' + 1 \leq 0)$
Induction Step
   G5.2.2 $[n' + m' + 1 \leq \text{length } l' \supset \text{rotate } L n'(\text{rotate } L m' + 1 l') = \text{rotate } L(n' + m' + 1)l']$
   $\supset$
   $[n' + m' + 1 \leq \text{length } (a :: l') \supset \text{rotate } L n'(\text{rotate } L m' + 1 (a :: l')) = \text{rotate } L(n' + m' + 1)(a :: l')]$
A4.2.2 $n' + m' + 1 \leq \text{length } l' \supset \text{rotate } L n'(\text{rotate } L m' + 1 l') = \text{rotate } L(n' + m' + 1)l'$
   G6.2.2 $n' + m' + 1 \leq \text{length } (a :: l') \supset \text{rotate } L n'(\text{rotate } L m' + 1 (a :: l')) = \text{rotate } L(n' + m' + 1)(a :: l')$
A5.2.2 $n' + m' + 1 \leq \text{length } (a :: l')$
   G7.2.2 $\text{rotate } L n'(\text{rotate } L m' + 1 (a :: l')) = \text{rotate } L(n' + m' + 1)(a :: l')$
A6.2.2 $n' + m' \leq \text{length } l'$
   A7.2.2 $m' \leq \text{length } l'$
   A8.2.2 $\text{rotate } L \ (m' + 1) (a :: l') = \text{rotate } L m'(l' @ [a])$
   A9.2.2 $\text{rotate } L (n' + m' + 1)(a :: l') = \text{rotate } L (n' + m')(l' @ [a])$
   G8.2.2 $\text{rotate } L n'(\text{rotate } L m' (l' @ [a])) = \text{rotate } L (n' + m')(l' @ [a])$
A10.2.2 From A3.2 and A6.2.2
   $\theta = \{ n \leftarrow n', l \leftarrow l' @ [a]\}$
   $\text{rotate } L n'(\text{rotate } L m' (l' @ [a])) = \text{rotate } L(n' + m')(l' @ [a])$
   G9.2.2 true
Problem 18

\begin{align*}
A_0. & \quad x = x \\
A_1. & \quad \text{rotate}_L (\text{length } l) l = l \\
A_2. & \quad \text{rotate}_R n l = \text{rotate}_L ((\text{length } l) - n) l \\
A_3. & \quad n + m \leq \text{length } l \supset \text{rotate}_L n (\text{rotate}_L m l) = \text{rotate}_L (n + m) l \\
& \quad \forall n \leq \text{length } l \supset (\text{rotate}_R n (\text{rotate}_L n l) = l) \\
& \quad G_2. (n' \leq \text{length } l') \supset (\text{rotate}_R n' (\text{rotate}_L n' l') = l') \\
A_4. & \quad n' \leq \text{length } l' \\
& \quad G_3. \text{rotate}_R n' (\text{rotate}_L n' l') = l' \\
& \quad G_4. \text{rotate}_L ((\text{length } l') - n') (\text{rotate}_L n' l') = l' \\
A_5. & \quad (\text{length } l') - n' \leq \text{length } l' \\
A_6. & \quad ((\text{length } l') - n') + n' \leq \text{length } l' \\
A_7. & \quad \text{rotate}_L ((\text{length } l') - n') (\text{rotate}_L n' l') = \text{rotate}_L ((\text{length } l') - n' + n') l' \\
A_8. & \quad \text{rotate}_L ((\text{length } l') - n') (\text{rotate}_L n' l') = \text{rotate}_L (\text{length } l') l' \\
A_9. & \quad \text{rotate}_L ((\text{length } l') - n') (\text{rotate}_L n' l') = l' \\
& \quad G_5. \text{true}
\end{align*}

References

