Permutation Module

smlhelp.github.io – Auxiliary Library

Jacob Neumann

June 2021

Contents

1 Definitions

Defn. 1 (Comparison Function)

For any type `t`, a value `cmp : t * t -> order` is a **comparison function** if it satisfies the usual properties that we expect from a function which consistently compares elements of type `t`:

1. `cmp` is total
2. **Reflexivity of `cmp`-equality:** For all `x:t`, `cmp(x,x)` $\cong$ `EQUAL`
3. **Symmetry of `cmp`-equality:** For all `x,y:t`, if `cmp(x,y)` $\cong$ `EQUAL`, then `cmp(y,z)` $\cong$ `EQUAL`
4. **Transitivity of `cmp`-equality:** For all `x,y,z:t`, if `cmp(x,y)` $\cong$ `EQUAL` and

cmp(y,z) ≅ EQUAL, then cmp(x,z) ≅ EQUAL

5.

Defn. 2 (Length)

- The **length** of `[]` is 0
- The length of `x::xs` is 1 plus the length of `xs`

Defn. 3 (List Membership)

For any type `t` and any comparison function `cmp : t ord`, a value `y : t` is **in** a list `L : t list` if `L = x::xs` where either

- `cmp(x,y)` ≅ `EQUAL`, or
- `y` is in `xs`.

We may denote this with `y` ∈ `L` or `y` $\in_{\texttt{cmp}}$ `L`.

Defn. 4 (Permutation)

Define the `count` function as follows:

```
fun count cmp (y,[]) = 0
  | count cmp (y,x::xs) =
      case (cmp(x,y)) of
        EQUAL => 1 + (count cmp (y,xs))
      | _ => count cmp (y,xs)
```

Given a type `t`, a comparison function `cmp : t * t -> order`, and two lists `L1,L2 : t list`, we say that `L1` is a **permutation** of `L2` (with respect to `cmp`) if

`count cmp (x,L1)` ≅ `count cmp (x,L2)` for all values `x:t`.

We omit mention of `cmp` if one is clear from context.[a]

[a] E.g. `Int.compare` when deciding whether one `int list` is a permutation of another.

Defn. 5 (Permutation Function)

A function `f : t list -> t list` is said to be a **permutation function** if `f` is total and for all values `L : t list`, the list `f(L)` is a permutation of `L`.

Defn. 6 (Splitting Function)

A function `s : t list -> t list * t list` is said to be a **splitting function** if it is total and

`op@(s(L))` is a permutation of `L`   for all values `L:t list`.

Or, in other words, if `A@B` is a permutation of `L`, where `(A,B)=s(L)`.

Defn. 7 (Merging Function)

A function `m : t list * t list -> t list` is said to be a **merging function** if it is total and

`m(A,B)` is a permutation of `A@B`   for all values `A,B:t list`.

Defn. 8 (Sorted)

A value `L : t list` is **sorted** (with respect to `cmp : t ord`) if either:

- `L = []`
- `L = [x]` for some `x`
- `L = x::x'::xs` for some `x,x',xs` such that
  - `cmp(x,x')` evaluates to either `LESS` or `EQUAL`
  - `x'::xs` is sorted (with respect to `cmp`)

2 Specification

2.1 Types

Type Spec

```
type 'a ord = 'a * 'a -> order
```

INVARIANT: Any value `cmp : t ord` is a comparison function

Type Spec

```
type 'a perm = 'a list -> 'a list
```

INVARIANT: Any value `f : t perm` is a permutation function

Type Spec

```
type 'a splitter = 'a -> 'a list * 'a list
```

INVARIANT: Any value `s : 'a ord` is a splitting function

Type Spec

```
type 'a merger = 'a list * 'a list -> 'a list
```

INVARIANT: Any value `m : 'a merger` is a merging function

2.2 Values

Value Spec

```
rev : 'a perm
```

REQUIRES: true

ENSURES: `rev L` evaluates to a list containing the elements of `L`, in the reverse order

WORK: $O(n)$, where n is the length of the input list

SPAN: $O(n)$

Value Spec

```
riffle : 'a perm
```

REQUIRES: true

ENSURES: `riffle L` evaluates to a "riffle shuffle" permutation of `L`: the first half of the list interleaved with the second half

Value Spec

```
cleanSplit : 'a splitter
```

REQUIRES: true

ENSURES: `cleanSplit(L)` $\Longrightarrow$ `(A,B)` where the lengths of `A` and `B` differ by at most one, and `L` $\cong$ `A@B`

Value Spec

```
split : 'a splitter
```

REQUIRES: true

ENSURES: `split(L)` $\Longrightarrow$ `(A,B)` where the lengths of `A` and `B` differ by at most one

WORK: $O(n)$

SPAN: $O(n)$

Value Spec

```
interleave : 'a merger
```

REQUIRES: true

ENSURES: `interleave(A,B)` consists of alternating elements of `A` and of `B`, in the same order they were in in their respective input lists

Value Spec

`merge : 'a ord -> 'a merger`

REQUIRES: `A` and `B` are sorted with respect to `cmp`

ENSURES: `merge cmp (A,B)` is sorted with respect to `cmp`

WORK: `merge cmp (A,B)` is $O(m+n)$, where m and n are the lengths of `A` and `B`, respectively (this assumes `cmp` is $O(1)$)

SPAN: $O(m+n)$

Value Spec

`msort : 'a ord -> 'a perm`

REQUIRES: true

ENSURES: `msort cmp L` evaluates to a sorted (w.r.t `cmp`) permutation of `L`

WORK: `msort cmp L` is $O(n \log n)$ where n is the length of `L` (assuming `cmp` is $O(1)$)

SPAN: $O(n)$

3 Lemmas

Lemma 9

For all types `t` and all values `X,Y,Z : t list`,

$$\texttt{(X @ Y) @ Z} \quad \cong \quad \texttt{X @ (Y @ Z)}$$

Fact 10

For all types `t`, all values `x : t` and all values `L : t list`,

$$\texttt{[x]@L} \quad \cong \quad \texttt{x::L}$$

Fact 11

For all types `t` and all values `L : t list`,

$$\texttt{L@[]} \quad \cong \quad \texttt{L}$$

Lemma 12

For all types `t`, all `cmp : t ord`, all values `y : t` and all values `L1,L2 : t list`,

`count cmp (y,L1@L2)` $\cong$ `(count cmp (y,L1)) + (count cmp (y,L2))`

4 rev Correctness and Analysis

4.1 Behavior

The implementation of `rev` given in `Permute.sml` is as follows.

aux-library: Permute.sml

```
local
  fun trev ([],acc) = acc
    | trev (x::xs,acc) = trev(xs,x::acc)
in
  val rev = fn L => trev(L,[])
end
```

We must prove that this (a) indeed defines a value of type `'a perm`, (b) that `rev` reverses its input list, and then (c) analyze its runtime. We'll start with (b), by proving `rev` equivalent to a more canonical version of list reverse.

For the sake of this document, we'll understand the meaning of "reverse order" (as it appears in the spec of `rev`) to be given by the following function.

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs)@[x]
```

which happens to be total:

Fact 13

`reverse` is total

So to prove the ENSURES of `rev`, we'll prove that `rev` and `reverse` are extensionally equivalent as functions. We begin with the following lemma about `trev`, the helper function for `rev`.

Lemma 14

For all types `t`, and all values `L : t list` and `acc : t list`

`trev(L,acc)` $\cong$ `(reverse L)@acc`.

Proof. By structural induction on `L`.

BC `L=[]`. Let `acc` be arbitrary.

```
trev([],acc) ≅ acc                      (Defn. trev)
             ≅ []@acc                   (Defn. @)
             ≅ (reverse [])@acc         (Defn. reverse)
```

IS `L=x::xs` for some values `x : t`, `xs : t list`

IH `trev(xs,acc') ≅ (reverse xs)@acc'` for all values `acc' : t list`

Let `acc : t list` be arbitrary. *WTS:*

```
trev(x::xs,acc)  ≅  (reverse (x::xs))@acc
```

```
trev(x::xs,acc) ≅ trev(xs,x::acc)            (Defn. trev)
                ≅ (reverse xs)@(x::acc)       IH
                ≅ ((reverse xs)@[x])@acc     (Fact 10, Fact 13, Lemma 9)
                ≅ reverse(x::xs) @ acc       (Defn. reverse)
```

Done. □

Then the correctness of `rev` is immediate:

Prop 15

`rev ≅ reverse`

Proof. Pick an arbitrary type `t` and an arbitrary value `L : t list`. Then,

```
rev L ≅ trev(L,[])              (Defn. rev)
      ≅ (reverse L)@[]          (Lemma 14)
      ≅ reverse L               (Fact 13, Fact 11)
```

proving the claim. □

Cor. 16

`rev` is total.

4.2 Asymptotic Complexity

5 `cleanSplit`, `interleave`, and `riffle`

6 Sorting