# The Nuprl Theory of Lists

Radhika Lakshmanan

August 5, 2003

**Abstract**

In addition to the standard list theory, a variety of Nuprl users have added definitions and theorems about list operations to the Nuprl library. To make them more accessible, the library had to be reorganized and completed by adding theorems that were obviously missing. The new list library resulting from these efforts is described in this document.

## 1 Introduction

Dependencies between theorems are crucial to the Nuprl Library. When I first set out to reorganize and complete the list library, I had not yet realized that fact. I wanted to create a list library that would be both easy to navigate, for new users of the system, and functional, so users could prove theorems based on the dependencies within that library. I started with the reorganization, assuming that the dependencies would resolve themseleves. Unfortunately, when I had finished, I found that my new list library was not functional, due to the altered dependencies I had introduced. In addition, the various new theorems I had created also did not work with the altered dependencies. However, the system I came up with was easy to navigate, so I did not want to abandon it entirely. I therefore took Rich Eaton's suggestion and simply created two libraries with the same contents: one that would be used only for display purposes, so users would easily be able to see the contents of the library, and one with correct dependencies that users could employ to prove new theorems. These two libraries are the functional library and the display library. The functional library can be found in the directory '`list_code`'. The directory '`radhika presentation/lists`' is the display library.

Since the contents of the two libraries are identical, users can search the presentation library for any list theory they want, without having to look through the unorganized functional library. This enables users to more quickly access the information they may need to prove a theorem, or see if the information they require is missing from the library. Unfortunately, since there are two libraries, users must take care to update both when adding new theorems. However, it is obviously benficial to update both libraries, because it is convenient to have a easily navigable library available.

The following documentation describes the contents and the layout of the presentation library.

## 2 Map of the Presentation Library

The presentation library is fairly straightforward. The first three directories contain all the display forms, abstractions, and code objects, respectively, that are associated with the list library. The abstractions and display forms are simply in the order in which I originally found them in the standard `list` and `list+` libraries. The code objects have been organized, for the most part, by name. For example, all of the objects having to do with the filter function are grouped together.

Almost all of the rest of the directories' contents can be inferred from their names. With the exception of the 'length_thms' directory, there are almost no duplicate entries in this library, while there are many theorems which involve more than one operation. A user might expect the theorem map_append to be referenced in both the map directory and the append directory. This is not the case. Users may note that as one goes down the list of directories, the operations named by the directories tend to become more complicated. Since mapping is lower on the list than the append operation, it is considered to be more complex, so map_append is referenced in 'map_thms', not in 'append_thms'. Any theorem that involves more than one operation will be referenced in the more complicated operation's directory. Since proofs can only see what comes before them, in this way every proof will be able to see all its necessary components. Since the display library does not allow the user to prove theorems, the theorems are not actually required to be able to see their components, however, this style is consistent with the concepts of reference environents behind the Nuprl system, and so will help the user understand the system better. Here is a summary of the contents of each directory.

Note: Because this library is for presentation, many of the theorems in it have been linked under names which are different from their objects. The idea of this library is to help the user look up necessary theorems. Changing the links has made the library easier to navigate. However, the documentation contains the object names, not the link names, since object names must be used to reference a theorem in another proof.

- The 'list_defs' directory contains all the abstractions related to lists. The corresponding display forms have been moved to a separate directory list_dforms, which is listed at the end of this document. Tactics and other ML code related to lists is contained in the directory list_code.
- The 'null_and_basic_thms' directory contains theorems about null lists, list decomposition into head and tail, members of lists and how to determine the existence of members, and non-empty lists.
- The 'append_thms' directory contains theorems pertaining to the append operation.
- The 'map_thms' directory contains theorems pertaining to the map operation.
- The 'hd_tl_thms' directory contains theorems pertaining to heads and tails of lists. It also has the nth_tl theorems, and theorems associated with the last element operation.
- The 'reverse_thms' directory contains theorems pertaining to the reverse operation.
- The 'segment_thms' directory contains theorems pertaining to segments of lists, such as those associated with the first_n abstraction and the iseg abstraction. It also contains theorems associated with the safety abstraction.
- The 'select_reject_thms' directory contains theorems pertaining to the select operation, which selects the ith element of a list, and the reject operation, which deletes the ith element and returns the resulting list. This directory also contains the reject2 operation, which is the same as the reject operation, only defined non-recursively.
- The 'reduce_thms' directory contains theorems pertaining to the reduce, reduce2, for, and list_accum operations. The reduce operation is also known as a foldr operation. The list_accum operation is also known as a foldl operation. The for operation employs the reduce operation and the reduce2 operation works by using the element index.

- The 'listify_thms' directory contains the listify operation theorems, along with the mklist operation theorems. Both operations define methods of tabulating lists of a specified length n. The directory also contains properties pertaining to the length of these lists.
- The 'agree_on_common' directory contains theorems pertaining to the agree_on_common operation.
- The 'sublist_thms' directory contains theorems pertaining to sublists. This includes theorems about nil sublists and sublists made up of the head or tail of a list, along with disjoint sublists. It also contains theorems pertaining the l_before abstraction, which is related to sublists, the l_succ abstraction, and the l_subset abstraction. In addition, the theorems related to the sublist* abstraction are in this directory.
- The 'filter_thms' directory contains theorems pertaining to the filter and filter2 operations.
- The 'zip_unzip_thms' directory contains theorems pertaining to the zip and unzip operations.
- The 'all_exists_thms' directory contains theorems which assume that if a property is true for all elements of a list, or if there exists one element in a list for which a property is true, then some consequence can be proved. There are many varied theorems in this directory, so some browsing may be required to find something specific.
- The 'duplicates_thms' directory contains all theorems associate d with the no_repeats operation. It contains a few interesting theorems that I was not able to prove, with explanations as to why I was unable to prove them.
- The 'split_thms' directory contains theorems pertaining to the split operation, and the split_tail operation.
- The 'interleaving_thms' directory contains theorems pertaining to the interleaving abstractions, interleaving, interleaving_occurence, interleaved_family_occurence, and interleaved_family.
- The 'causal_order_thms' directory contains theorems pertaining to the causal_order abstraction.
- The 'permute_swap_thms' directory contains theorems pertaining to the permute_list abstraction and the swap abstraction.
- The 'index_thms' directory contains thms associated with the count abstraction and the first index abstraction.
- The 'length_thms' directory contains all theorems associated with the length of lists, usually after performing some operation. Most of these theorems are duplicated in the other directories. The proof map_length is found in both 'map_thms' and 'length_thms'.

If any more abstractions are added which are distinct from those in the directories above, a new directory should be added to catalog the resulting theorems.

## 2.1 The directory list_defs

```
C  abstractions_com
                ==================
                LIST ABSTRACTIONS
                ==================
                This directory contains all of the abstractions from both the standard list
                library and the list+ library.  Related functions are grouped for easier
                reference, i.e. map, mapc, and mapcons, hd and tl, reduce and reduce2.

A  null         null(as) ==  case as of [] => tt | a::as' => ff esac
A  append       as @ bs ==
                  Y (λappend,as.case as of [] => bs | a::as' => a::(append as') esac) as
A  length       ||as|| ==
                  Y (λlength,as.case as of [] => 0 | a::as' => (length as') + 1 esac) as
A  map          map(f;as) ==  Y (λmap,as.case as of [] => [] | a::as' => f a::(map as') esac) as
A  mapc         mapc(f) ==
                  Y (λmapc,f,as.case as of [] => [] | a::as1 => f a::(mapc f as1) esac) f
A  mapcons      mapcons(f;as) ==
                  Y (λmapcons,as.case as of [] => [] | a::as' => f a as'::(mapcons as') esac) as
A  hd           hd(l) ==  rec-case(l) of [] => "?" | h::t => v.h
A  tl           tl(l) ==  rec-case(l) of [] => [] | h::t => v.t
A  nth_tl       nth_tl(n;as) ==
                  Y (λnth_tl,n,as.if n ≤z 0 then as else nth_tl (n - 1) tl(as) fi ) n as
A  reverse      rev(as) ==
                  Y (λreverse,as.case as of [] => [] | a::as' => (reverse as') @ (a::[]) esac) as
A  firstn       firstn(n;as) ==
                  Y
                  (λfirstn,n,as.
                    case as of
                      [] => []
                      a::as' => if 0 <z n then a::(firstn (n - 1) as') else [] fi
                    esac)
                  n as
A  segment      as[m..n⁻] ==  firstn(n - m;nth_tl(m;as))
A  select       l[i] ==  hd(nth_tl(i;l))
A  reverse_select
                reverse_select(l;n) ==  l[||l|| - n + 1]
A  reject       as[i] ==
                  Y
                  (λreject,i,as.
                    if i ≤z 0
                    then tl(as)
                    else case as of [] => [] | a'::as' => a'::(reject (i - 1) as') esac
                    fi )
                  i as
A  reject2      bs[i] ==  if i ≤z 0 then tl(bs) else firstn(i;bs) @ nth_tl(i + 1;bs) fi
A  reduce       reduce(f;k;as) ==
                  Y (λreduce,as.case as of [] => k | a::as' => f a (reduce as') esac) as
A  reduce2      reduce2(f;k;i;as) ==
                  Y
                  (λreduce2,i,as.
                    case as of [] => k | a::as' => f a i (reduce2 (i + 1) as') esac)
                  i as
```

```
A  for          For{T,op,id} x ∈ as. f[x] ==  reduce(op;id;map(λx:T. f[x];as))
A  for_hdtl     ForHdTl{A,f,k} h::t ∈ as. g[h; t] ==  reduce(f;k;mapcons(λh,t.g[h; t];as))
A  listify      listify(f;m;n) ==
                  Y (λlistify,m.if n ≤z m then [] else f m::(listify (m + 1)) fi ) m
A  list_n       A List(n) ==  {x:A List| ||x|| = n}
A  mklist       mklist(n;f) ==  primrec(n;[];λi,l.(l @ (f i::[])))
A  l_member     (x ∈ l) ==  ∃i:ℕ. ((i < ||l||) c∧ (x = l[i]))
A  l_member!    (x ∈! l) ==
                  ∃i:
                   ((i < ||l||)
                   c∧ ((x = l[i]) ∧ (∀j:ℕ. ((j < ||l||) ⇒ (x = l[j]) ⇒ (j = i)))))
A  agree_on_common
                agree_on_common(T;as;bs) ==
                  Y
                  (λagree_on_common,as,bs.
                    case as of
                      [] => True
                      a::as' => case bs of
                                  [] => True
                                  b::bs' => ((¬(a ∈ bs)) ∧ (agree_on_common as' bs))
                                            ∨ ((¬(b ∈ as)) ∧ (agree_on_common as bs'))
                                            ∨ ((a = b) ∧ (agree_on_common as' bs'))
                                esac
                    esac)
                  as bs
A  agree_on     agree_on(T;x.P[x]) ==
                  λL1,L2.
                   ((||L1|| = ||L2||)
                   c∧ (∀i:ℕ||L1||. ((P[L1[i]] ∨ P[L2[i]]) ⇒ (L1[i] = L2[i]))))
A  last         last(L) ==  L[||L|| - 1]
A  sublist      L1 ⊆ L2 ==
                  ∃f:ℕ||L1|| → ℕ||L2||
                   (increasing(f;||L1||) ∧ (∀j:ℕ||L1||. (L1[j] = L2[f j])))
A  sublist_occurence
                sublist_occurence(T;L1;L2;f) ==
                  increasing(f;||L1||) ∧ (∀j:ℕ||L1||. (L1[j] = L2[f j]))
A  l_subset     l_subset(T;as;bs) ==  ∀x:T. ((x ∈ as) ⇒ (x ∈ bs))
A  sublist*     sublist*(T;as;bs) ==  ∀cs:T List. (cs ⊆ as ⇒ l_subset(T;cs;bs) ⇒ cs ⊆ bs)
A  disjoint_sublists
                disjoint_sublists(T;L1;L2;L) ==
                  ∃f1:ℕ||L1|| → ℕ||L||
                   ∃f2:ℕ||L2|| → ℕ||L||
                    ((increasing(f1;||L1||) ∧ (∀j:ℕ||L1||. (L1[j] = L[f1 j])))
                    ∧ (increasing(f2;||L2||) ∧ (∀j:ℕ||L2||. (L2[j] = L[f2 j])))
                    ∧ (∀j1:ℕ||L1||. ∀j2:ℕ||L2||.  (¬((f1 j1) = (f2 j2)))))
A  l_before     x before y ∈ l ==  x::y::[] ⊆ l
A  strong_before
                x << y ∈ l ==
                  ((x ∈ l) ∧ (y ∈ l))
                  ∧ (∀i,j:ℕ.
                       ((i < ||l||) ⇒ (j < ||l||) ⇒ (l[i] = x) ⇒ (l[j] = y) ⇒ (i < j)))
A  same_order   same_order(x1;y1;x2;y2;L;T) ==
                  x1 << y1 ∈ L ⇒ (x2 ∈ L) ⇒ (y2 ∈ L) ⇒ x2 << y2 ∈ L
```

```
A  l_succ        y = succ(x) in l
                 ⇒ P[y] ==  ∀i:ℕ. (((i + 1) < ||l||) ⇒ (l[i] = x) ⇒ P[l[i + 1]])
A  listp         A List+ ==  {l:A List| ↑0 <z ||l||}
A  count         count(P;L) ==  reduce(λa,n.(if P a then 1 else 0 fi  + n);0;L)
A  filter        filter(P;l) ==  reduce(λa,v.if P a then a::v else v fi ;[];l)
A  filter2       filter2(P;L) ==  reduce2(λx,i,l.if P i then x::l else l fi ;[];0;L)
A  iseg          l1 ≤ l2 ==  ∃l:T List. (l2 = (l1 @ l))
A  compat        l1 || l2 ==  l1 ≤ l2 ∨ l2 ≤ l1
A  list_accum    list_accum(x,a.f[x; a];y;l) ==
                   Y (λlist_accum,y,l.case l of [] => y | b::l' => list_accum f[y;b] l' esac) y l
A  zip           zip(as;bs) ==
                   Y
                   (λzip,as,bs.
                     case as of
                       [] => []
                       a::as' => case bs of [] => [] | b::bs' => <a, b>::(zip as' bs') esac
                     esac)
                   as bs
A  unzip         unzip(as) ==  <map(λp.(p.1);as), map(λp.(p.2);as)>
A  find          (first x ∈ as s.t. P[x] else d) ==
                   case filter(λx.P[x];as) of [] => d | a::b => a esac
A  list_all      list_all(x.P[x];l) ==  reduce(λa,b.(P[a] ∧ b);True;l)
A  l_all         (∀x∈L.P[x]) ==  ∀x:T. ((x ∈ L) ⇒ P[x])
A  l_all2        (∀x<y∈L.P[x; y]) ==  ∀x,y:T.  (x before y ∈ L ⇒ P[x; y])
A  l_all_since   (∀x≥a∈L.P[x]) ==  P[a] ∧ (∀b:T. (a before b ∈ L ⇒ P[b]))
A  l_exists      (∃x∈L.P[x]) ==  ∃x:T. ((x ∈ L) ∧ P[x])
A  no_repeats    no_repeats(T;l) ==
                   ∀i,j:ℕ.  ((i < ||l||) ⇒ (j < ||l||) ⇒ (¬(i = j)) ⇒ (¬(l[i] = l[j])))
A  l_disjoint    l_disjoint(T;l1;l2) ==  ∀x:T. (¬((x ∈ l1) ∧ (x ∈ l2)))
A  append_rel    append_rel(T;L1;L2;L) ==  (L1 @ L2) = L
A  safety        safety(A;tr.P[tr]) ==  ∀tr1,tr2:A List.  (tr1 ≤ tr2 ⇒ P[tr2] ⇒ P[tr1])
A  strong_safety
                 strong_safety(T;tr.P[tr]) ==  ∀tr1,tr2:T List.  (tr1 ⊆ tr2 ⇒ P[tr2] ⇒ P[tr1])
A  mapfilter     mapfilter(f;P;L) ==  map(f;filter(P;L))
A  split_tail    split_tail(L | ∀x.f[x]) ==
                   Y
                   (λsplit_tail,L.
                     case L of
                       [] => <[], []>
                       a::as => let <hs,ftail> = split_tail as
                                in
                                case hs of
                                  [] => if f[a] then <[], a::ftail> else <a::[], ftail> fi
                                  x::y => <a::hs, ftail>
                                esac
                     esac)
                   L
A  interleaving  interleaving(T;L1;L2;L) ==
                   (||L|| = (||L1|| + ||L2||)) ∧ disjoint_sublists(T;L1;L2;L)
```

```
A  interleaving_occurence
                interleaving_occurence(T;L1;L2;L;f1;f2) ==
                  (||L|| = (||L1|| + ||L2||))
                  ∧ (increasing(f1;||L1||) ∧ (∀j:ℕ||L1||. (L1[j] = L[f1 j])))
                  ∧ (increasing(f2;||L2||) ∧ (∀j:ℕ||L2||. (L2[j] = L[f2 j])))
                  ∧ (∀j1:ℕ||L1||. ∀j2:ℕ||L2||.  (¬((f1 j1) = (f2 j2))))
A  interleaved_family_occurence
                interleaved_family_occurence(T;I;L;L2;f) ==
                  ((∀i:I. (increasing(f i;||L i||) ∧ (∀j:ℕ||L i||. (L i[j] = L2[f i j]))))
                  ∧ (∀i1,i2:I.
                       ((¬(i1 = i2))
                       ⇒ (∀j1:ℕ||L i1||. ∀j2:ℕ||L i2||.  (¬((f i1 j1) = (f i2 j2)))))))
                  ∧ (∀x:ℕ||L2||. ∃i:I. ∃j:ℕ||L i||. (x = (f i j)))
A  interleaved_family
                interleaved_family(T;I;L;L2) ==
                  ∃f:i:I → ℕ||L i|| → ℕ||L2||. interleaved_family_occurence(T;I;L;L2;f)
A  causal_order causal_order(L;R;P;Q) ==
                  ∀i:ℕ||L||. ((Q i) ⇒ (∃j:ℕ||L||. (((j ≤ i) ∧ (P j)) ∧ (R j i))))
A  permute_list (L o f) ==  mklist(||L||;λi.L[f i])
A  swap         swap(L;i;j) ==  (L o (i, j))
A  guarded_permutation
                guarded_permutation(T;P) ==
                  (λL1,L2.∃i:ℕ||L1|| - 1. ((P L1 i) ∧ (L2 = swap(L1;i;i + 1))))^*
A  count_index_pairs
                count(i<j<||L|| : P L i j) ==
                  sum(if i <z j ∧_b (P L i j) then 1 else 0 fi  | i < ||L||; j < ||L||)
A  count_pairs  count(x < y in L | P[x; y]) ==
                  sum(if i <z j ∧_b P[L[i]; L[j]] then 1 else 0 fi  | i < ||L||; j < ||L||)
A  first_index  index-of-first x in L.P[x] ==  search(||L||;λi.P[L[i]])

END  list_defs
```

## 2.2 The directory null_and_basic_thms

C stdcomm

```
==================================
NULL THEOREMS and BASIC DEFINITIONS
==================================
This directory contains theories related to the basic list definitions such as
null, list member, and non-empty lists, and also contains basic list
decomposition theorems.
```

| | | |
|---|---|---|
| S | null_wf | ∀T:𝕌. ∀as:T List. (null(as) ∈ 𝔹) |
| S | null_wf2 | null([]) ∈ |
| S | assert_of_null | ∀T:𝕌. ∀as:T List. (↑null(as) ⇔ as = []) |
| S | length_nil | \|\|[]\|\| = 0 |
| S | length_of_null_list | ∀A:𝕌. ∀as:A List. ((as = []) ⇒ (\|\|as\|\| = 0)) |
| S | length_zero | ∀T:𝕌. ∀l:T List. (\|\|l\|\| = 0 ⇔ l = []) |
| S | length_of_not_nil | ∀A:𝕌. ∀as:A List. (¬(as = []) ⇔ \|\|as\|\| ≥ 1 ) |
| S | non_nil_length | ∀T:𝕌. ∀L:T List. ((¬(L = [])) ⇒ (0 < \|\|L\|\|)) |
| S | singleton_length | ∀T:𝕌. ∀x:T. (\|\|x::[]\|\| = 1) |
| S | length_cons | ∀A:𝕌. ∀a:A. ∀as:A List. (\|\|a::as\|\| = (\|\|as\|\| + 1)) |
| S | hd_wf | ∀A:𝕌. ∀l:A List. ((\|\|l\|\| ≥ 1 ) ⇒ (hd(l) ∈ A)) |
| S | tl_wf | ∀A:𝕌. ∀l:A List. (tl(l) ∈ A List) |
| S | list_decomp | ∀T:𝕌. ∀L:T List. ((0 < \|\|L\|\|) ⇒ (L ≜ hd(L)::tl(L))) |
| S | list_decomp_reverse | ∀T:𝕌. ∀L:T List. ((0 < \|\|L\|\|) ⇒ (∃x:T. ∃L':T List. (L = (L' @ (x::[]))))) |
| S | list_2_decomp | ∀T:𝕌. ∀z:T List. ((\|\|z\|\| = 2) ⇒ (z = (z[0]::z[1]::[]))) |
| S | l_member_wf | ∀T:𝕌. ∀x:T. ∀l:T List. ((x ∈ l) ∈ ℙ) |
| S | comb_for_l_member_wf | λT,x,l,z.(x ∈ l) ∈ T:𝕌 → x:T → l:T List → ↓True → |
| S | member_exists | ∀T:𝕌. ∀L:T List. ((¬(L = [])) ⇒ (∃x:T. (x ∈ L))) |
| S | member_tl | ∀T:𝕌. ∀as:T List. ∀x:T. ((0 < \|\|as\|\|) ⇒ (x ∈ tl(as)) ⇒ (x ∈ as)) |
| S | l_member_hd | ∀T:𝕌. ∀L:T List. ∀x:T. ((0 < \|\|L\|\|) ⇒ (x = hd(L)) ⇒ (x ∈ L)) |
| S | l_member_hd_tl | ∀T:𝕌. ∀L:T List. ∀x:T. ((x ∈ L) ⇒ ((x = hd(L)) ∨ (x ∈ tl(L)))) |
| S | nil_member | ∀T:𝕌. ∀x:T. ((x ∈ []) ⇔ False) |
| S | null_member | ∀T:𝕌. ∀L:T List. ∀x:T. ((↑null(L)) ⇒ (¬(x ∈ L))) |
| S | member_null | ∀T:𝕌. ∀L:T List. ∀x:T. ((x ∈ L) ⇒ (¬↑null(L))) |
| S | l_member_null3 | ∀T:𝕌. ∀L:T List. ∀x:T. ((L = []) ⇒ (¬(x ∈ L))) |
| S | l_member_non_nil | ∀T:𝕌. ∀x:T. ∀L:T List. ((x ∈ L) ⇒ (¬(L = []))) |
| S | cons_member | ∀T:𝕌. ∀l:T List. ∀a,x:T. ((x ∈ a::l) ⇔ (x = a) ∨ (x ∈ l)) |
| S | l_member_length | ∀T:𝕌. ∀L:T List. ((\|\|L\|\| ≥ 1 ) ⇒ (∃x:T. (x ∈ L))) |
| S | l_member_length_iff | ∀T:𝕌. ∀L:T List. (∃x:T. (x ∈ L) ⇔ \|\|L\|\| ≥ 1 ) |
| S | l_member_decidable | ∀T:𝕌. ∀x:T. ∀l:T List. ((∀y:T. Dec(x = y)) ⇒ Dec((x ∈ l))) |
| S | member_singleton | ∀T:𝕌. ∀a,x:T. ((x ∈ a::[]) ⇔ x = a) |
| S | l_member_hd_tl_iff | ∀T:𝕌. ∀L:T List. ∀x:T. ((¬(L = [])) ⇒ ((x ∈ L) ⇔ (x = hd(L)) ∨ (x ∈ tl(L)))) |
| S | l_member!_wf | ∀T:𝕌. ∀l:T List. ∀x:T. ((x ∈! l) ∈ ℙ) |
| S | cons_member! | ∀T:𝕌. ∀l:T List. ∀a,x:T. ((x ∈! a::l) ⇔ ((x = a) ∧ (¬(x ∈ l))) ∨ ((x ∈! l) ∧ (¬(x = a)))) |
| S | nil_member! | ∀T:𝕌. ∀x:T. ((x ∈! []) ⇔ False) |
| S | listp_wf | ∀A:𝕌. (A List$^+$ ∈ 𝕌) |
| S | listp_properties | ∀A:𝕌. ∀l:A List$^+$. (\|\|l\|\| ≥ 1 ) |
| S | hd_wf_listp | ∀A:𝕌. ∀l:A List$^+$. (hd(l) ∈ A) |
| S | comb_for_hd_wf_listp | λA,l,z.hd(l) ∈ A:𝕌 → l:A List$^+$ → ↓True → A |
| S | cons_wf_listp | ∀A:𝕌. ∀l:A List. ∀x:A. (x::l ∈ A List$^+$) |

```
S  comb_for_cons_wf_listp      λA,l,x,z.(x::l) ∈ A:𝕌 → l:A List → x:A → ↓True → A List
S  list_set_type               ∀T:𝕌. ∀L:T List. ∀P:T → ℙ.
                                        ((∀x∈L.P[x]) ⇒ (L ∈ {x:T| P[x]} List))
S  list_extensionality         ∀T:𝕌. ∀a,b:T List. ((||a|| = ||b||) ⇒
                                        (∀i:ℕ. ((i < ||a||) ⇒ (a[i] = b[i]))) ⇒ (a = b))
END  null_and_basic_thms
```

## 2.3 The directory append_thms

```
C  stdcomm      ================
                APPEND THEOREMS
                ================
                All theorems involving the append operation.

S  append_wf                    ∀T:𝕌. ∀as,bs:T List. (as @ bs ∈ T List)
S  comb_for_append_wf           λT,as,bs,z.(as @ bs) ∈ T:𝕌 → as:T List →
                                          bs:T List → ↓True → (T List)
S  append_assoc                 ∀T:𝕌. ∀as,bs,cs:T List. (((as @ bs) @ cs) = (as @ bs @ cs))
S  append_back_nil              ∀T:𝕌. ∀as:T List. ((as @ []) = as)
S  append_is_nil                ∀T:𝕌. ∀l1,l2:T List. ((l1 @ l2) = [] ⇔ (l1 = []) ∧ (l2 = []))
S  append_nil_sq                ∀T:𝕌. ∀l:T List. (l @ [] ≡ l)
S  null_append                  ∀T:𝕌. ∀L1,L2:T List. (null(L1 @ L2) ≡ null(L1) ∧b null(L2))
S  append_cons                  ∀T:𝕌. ∀L1,L2:T List. ∀a:T. (((a::L1) @ L2) = (a::(L1 @ L2)))
S  append_single_cons           ∀T:𝕌. ∀L:T List. ∀a:T. (((a::[]) @ L) = (a::L))
S  length_append                ∀T:𝕌. ∀as,bs:T List. (||as @ bs|| = (||as|| + ||bs||))
S  member_append                ∀T:𝕌. ∀x:T. ∀l1,l2:T List. ((x ∈ l1 @ l2)
                                          ⇔ (x ∈ l1) ∨ (x ∈ l2))
C  list_append_ind_com
                Alternative Induction Principle for Lists
                Used for multiset induction.

S  list_append_ind              ∀T:𝕌. ∀Q:T List → ℙ.
                                  (Q[[]]
                                  ⇒ (∀x:T. Q[x::[]])
                                  ⇒ (∀ys,ys':T List. (Q[ys] ⇒ Q[ys'] ⇒ Q[ys @ ys']))
                                  ⇒ {∀zs:T List. Q[zs]})

S  list_append_singleton_ind    ∀T:𝕌. ∀Q:T List → ℙ.
                                  (Q[[]]
                                  ⇒ (∀ys:T List. ∀x:T. (Q[ys] ⇒ Q[ys @ (x::[])]))
                                  ⇒ {∀zs:T List. Q[zs]})

S  append_rel_wf                ∀T:𝕌. ∀L1,L2,L:T List. (append_rel(T;L1;L2;L) ∈ ℙ)

END  append_thms
```

## 2.4 The directory map_thms

```
C  map_com      ==============
                MAP THEOREMS
                ==============
                All theorems pertaining to the map function. Some of these may be cross
                referenced in other directories.

S  map_wf                      ∀A,B:𝕌. ∀f:A → B. ∀l:A List. (map(f;l) ∈ B List)
S  comb_for_map_wf             λA,B,f,l,z.map(f;l) ∈ A:𝕌 → B:𝕌 → f:A
                                                  → B → l:A List → ↓True → (B List)
S  map_length                  ∀A,B:𝕌. ∀f:A → B. ∀as:A List. (||map(f;as)|| = ||as||)
S  map_length_nat              ∀A,B:𝕌. ∀f:A → B. ∀as:A List. (||map(f;as)|| = ||as||)
S  member_map                  ∀T,T':𝕌. ∀a:T List. ∀x:T'. ∀f:T → T'.
                                                  ((x ∈ map(f;a)) ⇔ ∃y:T. ((y ∈ a) ∧ (x = (f y))))
S  map_map                     ∀A,B,C:𝕌. ∀f:A → B. ∀g:B → C. ∀as:A List.
                                                  (map(g;map(f;as)) = map(g o f;as))
S  map_append                  ∀A,B:𝕌. ∀f:A → B. ∀as,as':A List.
                                                  (map(f;as @ as') = (map(f;as) @ map(f;as')))
S  map_append_sq               ∀f,as':Top. ∀A:𝕌. ∀as:A List.
                                                  (map(f;as @ as') ≐ map(f;as) @ map(f;as'))
S  map_id                      ∀A:𝕌. ∀as:A List. (map(Id{A};as) = as)
S  mapcons_wf                  ∀A,B:𝕌. ∀f:A → A List → B. ∀l:A List. (mapcons(f;l) ∈ B List)

C  mapc_com     ====================
                Curried map function
                ====================
                Illustration of use of add_rec_def for a curried function.

S  mapc_wf                     ∀A,B:𝕌. ∀f:A → B. (mapc(f) ∈ A List → (B List))
S  map_equal                   ∀T,T':𝕌. ∀a:T List. ∀f,g:T → T'.
                                                  ((∀i:ℕ. ((i < ||a||) ⇒ ((f a[i]) = (g a[i]))))
                                                  ⇒ (map(f;a) = map(g;a)))
S  map_equal2                  ∀T,T':𝕌. ∀a:T List. ∀f,g:T → T'.
                                                  ((∀x:T. ((x ∈ a) ⇒ ((f x) = (g x))))
                                                  ⇒ (map(f;a) = map(g;a)))
S  map_equal3                  ∀T,T':𝕌. ∀a:T List+. ∀f,g:T → T'.
                                                  ((∀x:T. ((x ∈ a) ⇒ ((f x) = (g x))))
                                                  ⇒ (map(f;a) = map(g;a)))
S  trivial_map                 ∀T:𝕌. ∀a:T List. ∀f:T → T.
                                                  ((∀x:T. ((x ∈ a) ⇒ ((f x) = x))) ⇒ (map(f;a) = a))
S  hd_map                      ∀T,T':𝕌. ∀a:T List+. ∀f:T → T'. (hd(map(f;a)) = (f hd(a)))
S  map_tl                      ∀A,B:𝕌. ∀L:A List. ∀f:A → B. (map(f;tl(L)) = tl(map(f;L)))
S  map_wf_listp                ∀A,B:𝕌. ∀f:A → B. ∀l:A List+. (map(f;l) ∈ B List+)

END  map_thms
```

## 2.5 The directory hd_tl_thms

```
C  hd_tl_com     ==================
                 HD and TL THEOREMS
                 ==================
                 All hd and tl theorems along with list decomposition theorems and theorems
                 having to do with the last element of a list.

S  hd_wf                    ∀A:𝕌. ∀l:A List. ((||l|| ≥ 1 ) ⇒ (hd(l) ∈ A))
S  hd_append                ∀T:𝕌. ∀L,L':T List. ((¬(L = [])) ⇒ (hd(L @ L') = hd(L)))
S  tl_wf                    ∀A:𝕌. ∀l:A List. (tl(l) ∈ A List)
S  length_tl                ∀A:𝕌. ∀l:A List. ((||l|| ≥ 1 ) ⇒ (||tl(l)|| = (||l|| - 1)))
S  tl_length2               ∀T:𝕌. ∀L:T List. ((¬(L = [])) ⇒ (||L|| = (||tl(L)|| + 1)))
S  tl_append                ∀T:𝕌. ∀L:T List. ∀x:T. ((¬(L = []))
                                 ⇒ (tl(L @ (x::[])) = (tl(L) @ (x::[]))))
S  nth_tl_wf                ∀A:𝕌. ∀as:A List. ∀i:ℤ. (nth_tl(i;as) ∈ A List)
S  nth_tl_decomp            ∀T:𝕌. ∀m:ℕ. ∀L:T List. ((m < ||L||)
                                 ⇒ (nth_tl(m;L) ≜ L[m]::nth_tl(1 + m;L)))
S  nth_tl_decomp_eq         ∀T:𝕌. ∀m:ℕ. ∀L:T List. ((m < ||L||)
                                 ⇒ (nth_tl(m;L) = (L[m]::nth_tl(1 + m;L))))
S  length_nth_tl            ∀A:𝕌. ∀as:A List. ∀n:{0...||as||}. (||nth_tl(n;as)|| = (||as|| - n))
S  comb_for_nth_tl_wf       λA,as,i,z.nth_tl(i;as) ∈ A:𝕌 → as:A List → i:ℤ
                                 → ↓True → (A List)
S  nth_tl_nil               ∀T:𝕌. ∀L:T List. ∀i:ℕ. ((i = ||L||) ⇒ (nth_tl(i;L) = []))
S  eq_cons_imp_eq_tls       ∀A:𝕌. ∀a,b:A. ∀as,bs:A List. (((a::as) = (b::bs)) ⇒ (as = bs))
S  eq_cons_imp_eq_hds       ∀A:𝕌. ∀a,b:A. ∀as,bs:A List. (((a::as) = (b::bs)) ⇒ (a = b))
S  cons_one_one             ∀T:𝕌. ∀a,a':T. ∀b,b':T List. ((a::b) = (a'::b')
                                 ⇔ {(a = a') ∧ (b = b')})
S  last_wf                  ∀T:𝕌. ∀L:T List. ((¬↑null(L)) ⇒ (last(L) ∈ T))
S  last_lemma               ∀T:𝕌. ∀L:T List. ((¬↑null(L))
                                 ⇒ (∃L':T List. (L = (L' @ (last(L)::[])))))
S  last_member              ∀T:𝕌. ∀L:T List. ((¬↑null(L)) ⇒ (last(L) ∈ L))
S  last_cons                ∀T:𝕌. ∀L:T List. ∀x:T. ((¬↑null(L)) ⇒ (last(x::L) = last(L)))
S  last_cons2               ∀T:𝕌. ∀L:T List. ∀a:T. ((¬(L = [])) ⇒ (last(a::L) = last(L)))
S  last_member2             ∀T:𝕌. ∀L:T List. ∀a,a':T. ((a' ∈ L @ (a::[]))
                                 ⇔ (a' = a) ∨ (a' ∈ L))
S  last_append_nil          ∀T:𝕌. ∀L,L':T List. ((L' = []) ⇒ (¬(L = []))
                                 ⇒ (last(L @ L') = last(L)))
S  last_append_not_nil      ∀T:𝕌. ∀L,L':T List. ((¬(L' = [])) ⇒ (last(L @ L') = last(L')))
S  last_singleton           ∀T:𝕌. ∀a:T. (last(a::[]) = a)
S  map_last                 ∀A,B:𝕌. ∀L:A List. ∀f:A → B.
                                 ((¬(L = [])) ⇒ (¬(map(f;L) = []))
                                 ⇒ (last(map(f;L)) = (f last(L))))

END  hd_tl_thms
```

## 2.6 The directory reverse_thms

```
C  reverse_com  =================
                REVERSE THEOREMS
                =================
                All theorems pertaining to the reverse operation.

S  reverse_wf                 ∀T:𝕌. ∀as:T List. (rev(as) ∈ T List)
S  reverse_null               ∀T:𝕌. ∀as:T List. ((as = []) ⇒ (rev(as) = as))
S  reverse_null_iff           ∀T:𝕌. ∀as:T List. (rev(as) = [] ⇔ as = [])
S  reverse_singleton          ∀T:𝕌. ∀a:T. (rev(a::[]) = (a::[]))
S  reverse_cons               ∀T:𝕌. ∀L:T List. ∀a:T. (rev(L @ (a::[])) = (a::rev(L)))
S  reverse_cons2              ∀T:𝕌. ∀L:T List. ∀a:T. (rev(a::L) = (rev(L) @ (a::[])))
S  reverse_append             ∀T:𝕌. ∀as,bs:T List. (rev(as @ bs) = (rev(bs) @ rev(as)))
S  rev_member                 ∀T:𝕌. ∀L:T List. ∀x:T. ((x ∈ rev(L)) ⇔ (x ∈ L))
S  rev_length                 ∀T:𝕌. ∀L:T List. (||L|| = ||rev(L)||)
S  reverse_reverse            ∀T:𝕌. ∀L:T List. (rev(rev(L)) = L)
S  reverse_map                ∀A,B:𝕌. ∀f:A → B. ∀L:A List. (map(f;rev(L)) = rev(map(f;L)))
S  reverse_if_then_else       ∀T:𝕌. ∀L,L':T List. ∀x:𝔹.
                                        (rev(if x then L else L' fi )
                                      = if x then rev(L) else rev(L') fi )
S  distribute_if              ∀T:𝕌. ∀L:T List. ∀x:T. ∀a:𝔹.
                                        (if a then L @ (x::[]) else L fi
                                      = (L @ if a then x::[] else [] fi ))

END  reverse_thms
```

## 2.7 The directory segment_thms

C segment_com ==========
SEGMENT THEOREMS
==========
All theorems involving segments of a list (not sublists), such as firstn, iseg, and safety.

S firstn_wf ∀A:𝕌. ∀as:A List. ∀n:ℤ. (firstn(n;as) ∈ A List)
S comb_for_firstn_wf λA,as,n,z.firstn(n;as) ∈ A:𝕌 → as:A List → n:ℤ → ↓True → (A List)
S append_firstn_lastn ∀T:𝕌. ∀L:T List. ∀n:{0...||L||}. ((firstn(n;L) @ nth_tl(n;L)) = L)
S length_firstn ∀A:𝕌. ∀as:A List. ∀n:{0...||as||}. (||firstn(n;as)|| = n)
S firstn_decomp ∀T:𝕌. ∀j:ℕ. ∀l:T List. ((0 < j) ⇒ (j < ||l||) ⇒ (firstn(j - 1;l) @ (l[j - 1]::[]) ≐ firstn(j;l)))
S firstn_all_length ∀T:𝕌. ∀L:T List. ∀i:ℕ. ((i = ||L||) ⇒ (firstn(i;L) = L))
S firstn_append ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ. ((i = ||L||) ⇒ (firstn(i;L @ (x::[])) = L))
S segment_wf ∀T:𝕌. ∀as:T List. ∀m,n:ℤ. (as[m..n⁻] ∈ T List)
S comb_for_segment_wf λT,as,m,n,z.(as[m..n⁻]) ∈ T:𝕌 → as:T List → m:ℤ → n:ℤ → ↓True → (T List)
S length_segment ∀T:𝕌. ∀as:T List. ∀i:{0...||as||}. ∀j:{i...||as||}. (||as[i..j⁻]|| = (j - i))
S iseg_wf ∀T:𝕌. ∀l1,l2:T List. (l1 ≤ l2 ∈ ℙ)
S comb_for_iseg_wf λT,l1,l2,z.l1 ≤ l2 ∈ T:𝕌 → l1:T List → l2:T List → ↓True →
S nil_iseg ∀T:𝕌. ∀l:T List. [] ≤ l
S iseg_nil2 ∀T:𝕌. ∀L:T List. (L ≤ [] ⇔ L = [])
S cons_iseg ∀T:𝕌. ∀a,b:T. ∀l1,l2:T List. (a::l1 ≤ b::l2 ⇔ (a = b) ∧ l1 ≤ l2)
S iseg_cons_one ∀T:𝕌. ∀L,L':T List. ∀x:T. ((¬(L' = [])) ⇒ (x::L ≤ L' ⇔ (x = hd(L')) ∧ L ≤ tl(L')))
S iseg_transitivity ∀T:𝕌. ∀l1,l2,l3:T List. (l1 ≤ l2 ⇒ l2 ≤ l3 ⇒ l1 ≤ l3)
S iseg_transitivity2 ∀T:𝕌. ∀l1,l2,l3:T List. (l2 ≤ l3 ⇒ l1 ≤ l2 ⇒ l1 ≤ l3)
S iseg_member ∀T:𝕌. ∀l1,l2:T List. ∀x:T. (l1 ≤ l2 ⇒ (x ∈ l1) ⇒ (x ∈ l2))
S iseg_append ∀T:𝕌. ∀l1,l2,l3:T List. (l1 ≤ l2 ⇒ l1 ≤ l2 @ l3)
S iseg_append0 ∀T:𝕌. ∀l1,l2:T List. l1 ≤ l1 @ l2
S iseg_append_nil ∀T:𝕌. ∀L,L':T List. (L @ L' ≤ L ⇒ (L' = []))
S iseg_extend ∀T:𝕌. ∀l1:T List. ∀v:T. ∀l2:T List. (l1 ≤ l2 ⇒ ((||l1|| < ||l2||) c∧ (l2[||l1||] = v)) ⇒ l1 @ (v::[]) ≤ l2)
S iseg_map ∀A,B:𝕌. ∀f:A → B. ∀L1,L2:A List. (L1 ≤ L2 ⇒ map(f;L1) ≤ map(f;L2))
S firstn_is_iseg ∀T:𝕌. ∀L1,L2:T List. (L1 ≤ L2 ⇔ ∃n:ℕ||L2|| + 1. (L1 = firstn(n;L2)))
S iseg_weakening ∀T:𝕌. ∀l:T List. l ≤ l
S iseg_length ∀T:𝕌. ∀l1,l2:T List. (l1 ≤ l2 ⇒ (||l1|| ≤ ||l2||))
S iseg_proper_length ∀T:𝕌. ∀L1,L2:T List. (L1 ≤ L2 ⇒ (||L1|| = ||L2||) ⇒ (L1 = L2))
S iseg_eq ∀T:𝕌. ∀L,L':T List. (L ≤ L' ⇒ L' ≤ L ⇒ (L = L'))
S compat_wf ∀T:𝕌. ∀l1,l2:T List. (l1 || l2 ∈ ℙ)
S common_iseg_compat ∀T:𝕌. ∀l,l1,l2:T List. (l1 ≤ l ⇒ l2 ≤ l ⇒ l1 || l2)
S safety_wf ∀A:𝕌. ∀P:A List → ℙ. (safety(A;x.P[x]) ∈ ℙ)

```
S  all_safety                ∀T,I:𝕌. ∀P:I → T List → ℙ.
                                      ((∀x:I. safety(T;L.P[x;L]))
                                      ⇒ safety(T;L.∀x:I. P[x;L]))
S  safety_and                ∀A:𝕌. ∀P,Q:A List → ℙ.
                                      (safety(A;x.P[x]) ⇒ safety(A;x.Q[x])
                                      ⇒ safety(A;x.P[x] ∧ Q[x]))
S  safety_nil                ∀T:𝕌. ∀P:T List → ℙ. ((∃l:T List. P[l])
                                      ⇒ safety(T;x.P[x]) ⇒ P[[]])
S  cond_safety_and           ∀A:𝕌. ∀P,Q:A List → ℙ.
                                      (safety(A;x.P[x]) ⇒ (∀tr1,tr2:A List. (tr1 ≤ tr2 ⇒
                                      P[tr2] ⇒ Q[tr2] ⇒ Q[tr1]))
                                      ⇒ safety(A;x.P[x] ∧ Q[x]))
S  safety_induced            ∀A,B:𝕌. ∀f:A → B. ∀P:B List → ℙ.
                                      (safety(B;L.P[L]) ⇒ safety(A;L.P[map(f;L)]))
S  strong_safety_wf          ∀A:𝕌. ∀P:A List → ℙ. (strong_safety(A;x.P[x]) ∈ ℙ)
S  strong_safety_safety      ∀A:𝕌. ∀P:A List → ℙ.
                                      (strong_safety(A;x.P[x]) ⇒ safety(A;x.P[x]))

END  segment_thms
```

## 2.8 The directory select_reject_thms

C select_reject_com

```
==========================
SELECT and REJECT THEOREMS
==========================
All theorems involving the select operation and the reject operation. Select
gives an element at a specific position in the list. Reject deletes the
element at the position specified and returns the resulting list. Reject2 is
the same operation as reject, but is defined non-recursively.
Note: The display forms for the reject and select abstractions are identical.
```

S select_wf ∀A:𝕌. ∀l:A List. ∀n:ℤ. ((0 ≤ n) ⇒ (n < ||l||) ⇒ (l[n] ∈ A))

S comb_for_select_wf λA,l,n,z.l[n] ∈ A:𝕌 → l:A List → n:ℤ → ↓(0 ≤ n) ∧ (n < ||l||) → A

S select_cons_hd ∀T:𝕌. ∀a:T. ∀as:T List. ∀i:ℤ. ((i ≤ 0) ⇒ (a::as[i] = a))

S select_cons_tl ∀T:𝕌. ∀a:T. ∀as:T List. ∀i:ℤ. ((0 < i) ⇒ (i ≤ ||as||) ⇒ (a::as[i] = as[i - 1]))

S select_cons_tl_sq2 ∀i:ℕ. ∀x,l:Top. (x::l[i + 1] ≏ l[i])

S select_cons ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℤ. ((i ≤ ||L||) ⇒ (x::L[i] = if i ≤z 0 then x else L[i - 1] fi ))

S select_member ∀T:𝕌. ∀L:T List. ∀i:ℕ||L||. (L[i] ∈ L)

S select_append_back ∀T:𝕌. ∀as,bs:T List. ∀i:{||as||..||as|| + ||bs||⁻}. (as @ bs[i] = bs[i - ||as||])

S select_append_front ∀T:𝕌. ∀as,bs:T List. ∀i:ℕ||as||. (as @ bs[i] = as[i])

S select_append ∀T:𝕌. ∀as,bs:T List. ∀i:ℕ||as|| + ||bs||. (as @ bs[i] = if i <z ||as|| then as[i] else bs[i - ||as||] fi )

S select_tl ∀A:𝕌. ∀as:A List. ∀n:ℕ||as|| - 1. (tl(as)[n] = as[n + 1])

S select_nth_tl ∀T:𝕌. ∀as:T List. ∀n:{0...||as||}. ∀i:ℕ||as|| - n. (nth_tl(n;as)[i] = as[i + n])

S select_firstn ∀T:𝕌. ∀as:T List. ∀n:{0...||as||}. ∀i:ℕn. (firstn(n;as)[i] = as[i])

S map_select ∀A,B:𝕌. ∀f:A → B. ∀as:A List. ∀n:ℕ||as||. (map(f;as)[n] = (f as[n]))

S select_equal ∀T:𝕌. ∀a,b:T List. ∀i:ℕ. ((a = b) ⇒ (i < ||a||) ⇒ (a[i] = b[i]))

S last_index ∀T:𝕌. ∀L:T List. ∀a:T. (L @ (a::[])[||L||] = a)

S reverse_select_wf ∀T:𝕌. ∀l:T List. ∀n:ℕ. ((n < ||l||) ⇒ (reverse_select(l;n) ∈ T))

S iseg_select ∀T:𝕌. ∀l1,l2:T List. (l1 ≤ l2 ⇔ (||l1|| ≤ ||l2||) c∧ (∀i:ℕ. ((i < ||l1||) ⇒ (l1[i] = l2[i]))))

S reverse_select ∀T:𝕌. ∀L:T List. ∀i:ℕ. ((i < ||L||) ⇒ (rev(L)[i] = L[||L|| - i + 1]))

S reject_wf ∀A:𝕌. ∀l:A List. ∀n:ℤ. (l[n] ∈ A List)

S reject_nil ∀T:𝕌. ∀L:T List. ∀i:ℕ. ((i < ||L||) ⇒ (¬(L = [])) ⇒ (L[i] = []) ⇒ (L = (L[i]::[])))

S reject_cons_hd ∀T:𝕌. ∀a:T. ∀as:T List. ∀i:ℤ. ((i ≤ 0) ⇒ ((a::as)[i] = as))

S reject_cons_hd2 ∀T:𝕌. ∀L:T List. ∀i:ℤ. ((i ≤ 0) ⇒ (¬(L = [])) ⇒ (L[i] = tl(L)))

```
S  reject_cons_tl           ∀T:𝕌. ∀a:T. ∀as:T List. ∀i:ℤ.
                                    ((0 < i) ⇒ (i ≤ ||as||)
                                    ⇒ ((a::as)[i] = (a::as[i - 1])))
S  reject_cons              ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ.
                                      ((i ≤ ||L||) ⇒ ((x::L)[i]
                                    = if i ≤z 0 then L else x::L[i - 1] fi ))
S  reject_equal             ∀T:𝕌. ∀as,bs:T List. ∀i:ℕ.
                                    ((as = bs) ⇒ (i < ||as||) ⇒ (¬(as = []))
                                    ⇒ (¬(bs = [])) ⇒ (as[i] = bs[i]))
S  reject_l_member          ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ.
                                    ((i < ||L||) ⇒ (¬(L = [])) ⇒ (x ∈ L)
                                    ⇒ (¬(x = L[i])) ⇒ (x ∈ L[i]))
S  reject_length            ∀T:𝕌. ∀L:T List. ∀i:ℕ.
                                    ((0 ≤ i) ⇒ (i < ||L||) ⇒ (¬(L = []))
                                    ⇒ (||L[i]|| = (||L|| - 1)))
S  reject_map               ∀A,B:𝕌. ∀cs:A List. ∀c:A. ∀i:ℤ. ∀f:A
                                      → B. ((i ≤ ||cs||) ⇒ (map(f;(c::cs)[i])
                                    = if i ≤z 0 then map(f;cs) else map(f;c::cs[i - 1]) fi))
S  reject2_wf               ∀T:𝕌. ∀L:T List. ∀i:ℕ. (L[i] ∈ T List)
S  reject2_nil              ∀T:𝕌. ∀L:T List. ∀i:ℕ.
                                     ((i < ||L||) ⇒ (¬(L = [])) ⇒ (L[i] = [])
                                     ⇒ (L = (L[i]::[])))
S  reject2_cons_hd          ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ. ((i ≤ 0)
                                    ⇒ ((x::L)[i] = L))
S  reject2_cons_tl          ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ.
                                    ((i < ||L||) ⇒ (i > 0) ⇒ ((x::L)[i] = (x::L[i - 1])))
S  reject2_append_singleton ∀T:𝕌. ∀L:T List. ∀x:T. ∀i:ℕ.
                                   ((¬(L = [])) ⇒ (i = ||L||)
                                   ⇒ ((L @ (x::[]))[i] = L))

END  select_reject_thms
```

## 2.9 The directory reduce_thms

```
C  reduce_com    ============
                 LIST FOLDING
                 ============
                 Reduce is the foldr operation and list_accum is the foldl operation. Reduce2 is
                 similar to reduce except that it works by index.

S  reduce_wf                  ∀A,B:𝕌. ∀f:A → B → B. ∀k:B. ∀as:A List. (reduce(f;k;as) ∈ B)
S  reduce_singleton           ∀A,B:𝕌. ∀f:A → B → B. ∀x:A. ∀acc:B.
                                  (reduce(f;acc;x::[]) = (f x acc))
S  for_wf                     ∀A,B,C:𝕌. ∀f:B → C → C. ∀k:C. ∀as:A List.
                                  ∀g:A → B. (For{A,f,k} x ∈ as. g[x] ∈ C)
S  for_hdtl_wf                ∀A,B,C:𝕌. ∀f:B → C → C. ∀k:C. ∀as:A List. ∀g:A
                                  → A List → B.
                                  (ForHdTl{A,f,k} h::t ∈ as. g[h;t] ∈ C)
S  comb_for_ifthenelse_wf     λb,A,p,q,z.if b then p else q fi  ∈ b:𝔹 → A:𝕌
                                  → p:A → q:A → ↓True → A
S  list_accum_wf              ∀T,T':𝕌. ∀l:T List. ∀y:T'. ∀f:T' → T
                                  → T'. (list_accum(x,a.f[x;a];y;l) ∈ T')
S  comb_for_list_accum_wf     λT,T',l,y,f,z.list_accum(x,a.f[x;a];y;l) ∈ T:
                                  → T': → l:T List
                                  → y:T' → f:T' → T → T'
                                  → ↓True → T'
S  list_accum_split           ∀T:𝕌. ∀i:ℕ. ∀l:T List. ∀f:Top → T
                                  → Top. ∀y:Top. ((i < ||l||)
                                  ⇒ (list_accum(x,a.f[x;a];y;l) ≙
                                  list_accum(x,a.f[x;a];
                                  list_accum(x,a.f[x;a];y;firstn(i;l));nth_tl(i;l))))
S  reduce2_wf                 ∀A,T:𝕌. ∀L:T List. ∀k:A. ∀i:ℕ. ∀f:T
                                  → {i..i + ||L||⁻} → A → A.
                                  (reduce2(f;k;i;L) ∈ A)

   .recall reduce2            obacc_recall{NIL:o, .recall reduce2:t, NIL:o, :t}
                                 (Obid: .recall reduce2;
                                  Obid: reduce2_df;
                                  Obid: reduce2;
                                  Obid: reduce2_wf)

S  reduce2_shift              ∀A,T:𝕌. ∀L:T List. ∀k:A. ∀i:ℕ. ∀f:T
                                  → {i..i + ||L||⁻} → A → A.
                                    (reduce2(f;k;i;L)
                                  = reduce2(λx,i,l.(f x (i - 1) l);k;i + 1;L))
S  comb_for_reduce2_wf        λA,T,L,k,i,f,z.reduce2(f;k;i;L) ∈ A:
                                  → T: → L:T List → k:A
                                  → i: → f:T → {i..i + ||L||⁻}
                                  → A → A → ↓True → A

END  reduce_thms
```

## 2.10 The directory listify_thms

```
C  listify_com  =================
                LISTIFY THEOREMS
                =================
                Theorems pertaining to the listify operation, the mklist operation, and the
                list_n property. Both the listify and the mklist operation are easy ways to
                tabulate lists.

S  listify_wf                ∀T:𝕌. ∀m,n:ℤ. ∀f:{m..n⁻} → T. (listify(f;m;n) ∈ T List)
S  comb_for_listify_wf       λT,m,n,f,z.listify(f;m;n) ∈ T: → m: → n:
                                                → f:{m..n⁻} → T → ↓True
                                                → (T List)
S  listify_length            ∀T:𝕌. ∀m,n:ℤ. ∀f:{m..n⁻} → T.
                                                ((n < m) ∨ (||listify(f;m;n)|| = (n - m)))
S  listify_select_id         ∀T:𝕌. ∀as:T List. ((λi:ℕ||as||. as[i])[ℕ||as||] = as)
S  list_n_wf                 ∀A:𝕌. ∀n:ℤ. (A List(n) ∈ 𝕌)
S  list_n_properties         ∀A:𝕌. ∀n:ℤ. ∀as:A List(n). (||as|| = n)
S  mklist_wf                 ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. (mklist(n;f) ∈ T List)
S  mklist_length             ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. (||mklist(n;f)|| = n)
S  mklist_select             ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. ∀i:ℕn. (mklist(n;f)[i] = (f i))

END  listify_thms
```

## 2.11 The directory agree_on_common_thms

```
C  agree_on_common_com
                ========================
                AGREE ON COMMON THEOREMS
                ========================
                Theorems pertaining to the agree_on_common operation.

S  agree_on_common_wf          ∀T:𝕌. ∀as,bs:T List. (agree_on_common(T;as;bs) ∈ ℙ)

   .recall agree_on_common     obacc_recall{NIL:o, .recall agree_on_common:t, NIL:o, :t}
                                           (Obid: .recall agree_on_common;
                                            Obid: agree_on_common_df;
                                            Obid: agree_on_common;
                                            Obid: agree_on_common_RecDef_additions;
                                            Obid: agree_on_common_wf)

S  agree_on_common_cons        ∀T:𝕌. ∀as,bs:T List. ∀x:T.
                                           (agree_on_common(T;x::as;x::bs)
                                           ⇔ agree_on_common(T;as;bs))
S  agree_on_common_cons2       ∀T:𝕌. ∀as,bs:T List. ∀x:T.
                                           (((¬(x ∈ bs)) ⇒ (agree_on_common(T;x::as;bs)
                                           ⇔ agree_on_common(T;as;bs)))
                                           ∧ ((¬(x ∈ as)) ⇒ (agree_on_common(T;as;x::bs)
                                           ⇔ agree_on_common(T;as;bs))))
S  agree_on_common_weakening   ∀T:𝕌. ∀as,bs:T List. ((as = bs) ⇒ agree_on_common(T;as;bs))
S  agree_on_common_symmetry    ∀T:𝕌. ∀as,bs:T List. (agree_on_common(T;as;bs)
                                           ⇒ agree_on_common(T;bs;as))
S  agree_on_common_nil         ∀T:𝕌. ∀as:T List. (agree_on_common(T;as;[]) ⇔ True)
S  agree_on_common_iseg        ∀T:𝕌. ∀as2,bs2,as1,bs1:T List. (as1 ≤ as2
                                           ⇒ bs1 ≤ bs2 ⇒ agree_on_common(T;as2;bs2)
                                           ⇒ agree_on_common(T;as1;bs1))
S  agree_on_common_append      ∀T:𝕌. ∀as,bs,cs,ds:T List.
                                           ((∀x∈cs.¬(x ∈ bs))
                                           ⇒ (∀x∈as.¬(x ∈ ds))
                                           ⇒ agree_on_common(T;as;cs)
                                           ⇒ agree_on_common(T;bs;ds)
                                           ⇒ agree_on_common(T;as @ bs;cs @ ds))
S  agree_on_wf                 ∀T:𝕌. ∀P:T → ℙ. (agree_on(T;a.P[a]) ∈ T List
                                           → T List → ℙ)
S  agree_on_equiv              ∀T:𝕌. ∀P:T → ℙ. EquivRel(T List)(_1 agree_on(T;a.P[a]) _2)

END  agree_on_common_thms
```

## 2.12 The directory sublist_thms

C sublist_com ==========
SUBLIST THEOREMS
==========
All sublist theorems including those pertaining to nil sublists and disjoint sublists. Also contains the l_before definition theorems, the l_succ theorems, and the l_subset theorems.

S sublist_wf ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ∈ ℙ)
S sublist_transitivity ∀T:𝕌. ∀L1,L2,L3:T List. (L1 ⊆ L2 ⇒ L2 ⊆ L3 ⇒ L1 ⊆ L3)
S length_sublist ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ (||L1|| ≤ ||L2||))
S proper_sublist_length ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ (||L1|| = ||L2||) ⇒ (L1 = L2))
S sublist_antisymmetry ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ L2 ⊆ L1 ⇒ (L1 = L2))
S member_sublist ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ {∀x:T. ((x ∈ L1) ⇒ (x ∈ L2))})
S member_iff_sublist ∀T:𝕌. ∀x:T. ∀L:T List. ((x ∈ L) ⇔ x::[] ⊆ L)
S sublist_append ∀T:𝕌. ∀L1,L2,L1',L2':T List. (L1 ⊆ L1' ⇒ L2 ⊆ L2'
⇒ L1 @ L2 ⊆ L1' @ L2')
S sublist_append1 ∀T:𝕌. ∀L1,L2:T List. L1 ⊆ L1 @ L2
S comb_for_sublist_wf λT,L1,L2,z.L1 ⊆ L2 ∈ T:𝕌 → L1:T List → L2:T List → ↓True →
S sublist_weakening ∀T:𝕌. ∀L1,L2:T List. ((L1 = L2) ⇒ L1 ⊆ L2)
S sublist_nil ∀T:𝕌. ∀L:T List. (L ⊆ [] ⇔ L = [])
S nil_sublist ∀T:𝕌. ∀L:T List. ([] ⊆ L ⇔ True)
S cons_sublist_nil ∀T:𝕌. ∀x:T. ∀L:T List. (x::L ⊆ [] ⇔ False)
S cons_sublist_cons ∀T:𝕌. ∀x1,x2:T. ∀L1,L2:T List.
(x1::L1 ⊆ x2::L2 ⇔ ((x1 = x2) ∧ L1 ⊆ L2)
∨ x1::L1 ⊆ L2)
S sublist_tl ∀T:𝕌. ∀L1,L2:T List. ((¬↑null(L2)) ⇒ L1 ⊆ tl(L2) ⇒ L1 ⊆ L2)
S sublist_tl2 ∀T:𝕌. ∀u:T. ∀v,L1:T List. (L1 ⊆ v ⇒ L1 ⊆ u::v)
S sublist_cons ∀T:𝕌. ∀L,L':T List. ∀x:T. (x::L' ⊆ L ⇒ L' ⊆ L)
S sublist_append_front ∀T:𝕌. ∀L,L1,L2:T List.
(((¬↑null(L)) ⇒ (¬(last(L) ∈ L2)))
⇒ L ⊆ L1 @ L2 ⇒ L ⊆ L1)
S sublist_pair ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. ((i < j) ⇒ L[i]::L[j]::[] ⊆ L)
S sublist_iseg ∀T:𝕌. ∀L1,L2:T List. (L1 ≤ L2 ⇒ L1 ⊆ L2)
S append_overlapping_sublists ∀T:𝕌. ∀L1,L2,L:T List. ∀x:T.
(no_repeats(T;L) ⇒ L1 @ (x::[]) ⊆ L ⇒ x::L2 ⊆ L
⇒ L1 @ (x::L2) ⊆ L)
S sublist_occurence_wf ∀T:𝕌. ∀L1,L2:T List. ∀f:ℕ||L1|| → ℕ||L2||.
(sublist_occurence(T;L1;L2;f) ∈ ℙ)
S range_sublist ∀T:𝕌. ∀L:T List. ∀n:ℕ. ∀f:ℕn
→ ℕ||L||. (increasing(f;n)
⇒ (∃L1:T List. ((||L1|| = n)
c∧ sublist_occurence(T;L1;L;f))))
S l_before_wf ∀T:𝕌. ∀l:T List. ∀x,y:T. (x before y ∈ l ∈ ℙ)
S l_before_append_front ∀T:𝕌. ∀L1,L2:T List. ∀x,y:T.
((¬(y ∈ L2)) ⇒ x before y ∈ L1 @ L2
⇒ x before y ∈ L1)
S weak_l_before_append_front ∀T:𝕌. ∀L1,L2:T List. ∀x,y:T.
((y ∈ L1) ⇒ (¬(y ∈ L2)) ⇒ x before y ∈ L1 @ L2
⇒ x before y ∈ L1)
S l_before_append ∀T:𝕌. ∀L1,L2:T List. ∀x,y:T. ((x ∈ L1) ⇒ (y ∈ L2)
⇒ x before y ∈ L1 @ L2)

```
S  l_before_member            ∀T:𝕌. ∀L:T List. ∀a,b:T. (a before b ∈ L ⇒ (b ∈ L))
S  l_before_member2           ∀T:𝕌. ∀L:T List. ∀a,b:T. (a before b ∈ L ⇒ (a ∈ L))
S  singleton_before           ∀T:𝕌. ∀a,x,y:T. (x before y ∈ a::[] ⇔ False)
S  nil_before                 ∀T:𝕌. ∀x,y:T. (x before y ∈ [] ⇔ False)
S  l_before_sublist           ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2
                                        ⇒ {∀x,y:T. (x before y ∈ L1
                                        ⇒ x before y ∈ L2)})
S  cons_before                ∀T:𝕌. ∀l:T List. ∀a,x,y:T.
                                        (x before y ∈ a::l ⇔ ((x = a) ∧ (y ∈ l))
                                        ∨ x before y ∈ l)
S  before_last                ∀T:𝕌. ∀L:T List. ∀x:T. ((x ∈ L) ⇒ (¬(x = last(L)))
                                        ⇒ x before last(L) ∈ L)
S  l_before_select            ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. ((j < i) ⇒ L[j] before L[i] ∈ L)
S  l_tricotomy                ∀T:𝕌. ∀x,y:T. ∀L:T List. ((x ∈ L) ⇒ (y ∈ L)
                                        ⇒ (((x = y) ∨ x before y ∈ L) ∨ y before x ∈ L))
S  strong_before_wf           ∀T:𝕌. ∀L:T List. ∀x,y:T. (x << y ∈ L ∈ ℙ)
S  same_order_wf              ∀T:𝕌. ∀L:T List. ∀x1,x2,y1,y2:T. (same_order(x1;y1;x2;y2;L;T) ∈ ℙ)
S  l_succ_wf                  ∀T:𝕌. ∀l:T List. ∀x:T. ∀P:T → ℙ. (y = succ(x) in l⇒ P[y] ∈ ℙ)
S  comb_for_l_succ_wf         λT,l,x,P,z.y = succ(x) in l⇒ P[y] ∈ T:
                                        → l:T List → x:T → P:T →
                                        → ↓True →
S  cons_succ                  ∀T:𝕌. ∀l:T List. ∀P:T → ℙ. ∀a,x:T.
                                        (y = succ(x) in a::l ⇒ P[y] ⇒ (((x = a)
                                        ⇒ (0 < ||l||) ⇒ P[hd(l)])
                                        ∧ ((¬(x = a)) ⇒ y = succ(x) in l⇒ P[y])))
S  l_before_transitivity      ∀T:𝕌. ∀l:T List. ∀x,y,z:T.
                                        (no_repeats(T;l) ⇒ x before y ∈ l ⇒ y before z ∈ l
                                        ⇒ x before z ∈ l)
S  l_before_antisymmetry      ∀T:𝕌. ∀l:T List. ∀x,y:T.
                                        (no_repeats(T;l) ⇒ x before y ∈ l
                                        ⇒ (¬y before x ∈ l))
S  disjoint_sublists_wf       ∀T:𝕌. ∀L1,L2,L:T List. (disjoint_sublists(T;L1;L2;L) ∈ ℙ)
S  disjoint_sublists_sublist  ∀T:𝕌. ∀L1,L2,L:T List. (disjoint_sublists(T;L1;L2;L)
                                         ⇒ {L1 ⊆ L ∧ L2 ⊆ L})
S  disjoint_sublists_witness  ∀T:𝕌. ∀L1,L2,L:T List.
                                        (disjoint_sublists(T;L1;L2;L)
                                        ⇒ (∃f:ℕ||L1|| + ||L2||
                                        → ℕ||L|| (Inj(ℕ||L1|| + ||L2||;ℕ||L||;f)
                                        ∧ (∀i:ℕ||L1|| + ||L2||
                                        (((i < ||L1||) ⇒ (L1[i] = L[f i]))
                                        ∧ ((||L1|| ≤ i) ⇒ (L2[i - ||L1||] = L[f i])))))))
S  length_disjoint_sublists   ∀T:𝕌. ∀L1,L2,L:T List.
                                        (disjoint_sublists(T;L1;L2;L)
                                        ⇒ ((||L1|| + ||L2||) ≤ ||L||))
S  l_before_iseg              ∀T:𝕌. ∀L1,L2:T List. ∀x,y:T. (L1 ≤ L2
                                        ⇒ x before y ∈ L1 ⇒ x before y ∈ L2)
S  l_subset_wf                ∀T:𝕌. ∀as,bs:T List. (l_subset(T;as;bs) ∈ ℙ)
S  sublist*_wf                ∀T:𝕌. ∀as,bs:T List. (sublist*(T;as;bs) ∈ ℙ)

END  sublist_thms
```

## 2.13 The directory filter_thms

C filter_com ==========
FILTER THEOREMS
==========
All filter and filter2 theorems. Filter uses the reduce operation, while filter2 uses the reduce2 operation.

S filter_wf ∀T:𝕌. ∀P:T → 𝔹. ∀l:T List. (filter(P;l) ∈ T List)
S length_filter ∀A:𝕌. ∀P:A → 𝔹. ∀L:A List. (||filter(P;L)|| = count(P;L))
S member_filter ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List.
∀x:T. ((x ∈ filter(P;L)) ⇔ (x ∈ L) ∧ (↑(P x)))
S filter_before ∀A:𝕌. ∀L:A List. ∀P:A → 𝔹. ∀x,y:A.
(x before y ∈ filter(P;L) ⇔ (↑(P x)) ∧
(↑(P y)) ∧ x before y ∈ L)
S agree_on_common_filter ∀T:𝕌. ∀P:T → 𝔹. ∀as,bs:T List.
(agree_on_common(T;as;bs)
⇒ agree_on_common(T;filter(P;as);filter(P;bs)))
S filter_functionality ∀A:𝕌. ∀L:A List. ∀f1,f2:A → 𝔹.
((f1 = f2) ⇒ (filter(f1;L) ≙ filter(f2;L)))
S filter_append ∀T:𝕌. ∀P:T → 𝔹. ∀l1,l2:T List.
(filter(P;l1 @ l2) ≙ filter(P;l1) @ filter(P;l2))
S filter_filter ∀T:𝕌. ∀P1,P2:T → 𝔹. ∀L:T List.
(filter(P1;filter(P2;L))
≙ filter(λt.((P1 t) ∧$_b$ (P2 t));L))
S filter_filter_reduce ∀T:𝕌. ∀P1,P2:T → 𝔹. ∀L:T List.
((∀x:T. ((↑(P1 x)) ⇒ (↑(P2 x))))
⇒ (filter(P1;filter(P2;L)) ≙ filter(P1;L)))
S filter_type ∀T:𝕌. ∀P:T → 𝔹. ∀l:T List. (filter(P;l) ∈ {x:T| ↑(P x)} List)
S filter_map ∀T1,T2:𝕌. ∀f:T1 → T2. ∀Q:T2
→ 𝔹. ∀L:T1 List.
(filter(Q;map(f;L)) = map(f;filter(Q o f;L)))
S mapfilter_wf ∀T:𝕌. ∀P:T → 𝔹. ∀T':𝕌. ∀f:{x:T| ↑(P x)}
→ T'. ∀L:T List.
(mapfilter(f;P;L) ∈ T' List)
S member_map_filter ∀T:𝕌. ∀P:T → 𝔹. ∀T':𝕌. ∀f:{x:T| ↑(P x)}
→ T'. ∀L:T List. ∀x:T'.
((x ∈ mapfilter(f;P;L)) ⇔ ∃y:T. ((y ∈ L)
∧ ((↑(P y)) c∧ (x = (f y)))))
S filter_iseg ∀T:𝕌. ∀P:T → 𝔹. ∀L2,L1:T List. (L1 ≤ L2
⇒ filter(P;L1) ≤ filter(P;L2))
S filter_safety ∀T:𝕌. ∀P:T List → ℙ. ∀f:T → 𝔹. (safety(T;L.P L)
⇒ safety(T;L.P filter(f;L)))
S filter_strong_safety ∀T:𝕌. ∀P:T List → ℙ. ∀f:T → 𝔹.
(strong_safety(T;L.P L)
⇒ strong_safety(T;L.P filter(f;L)))
S filter_trivial ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. ((∀x∈L.↑P[x])
⇒ (filter(P;L) ≙ L))
S filter_trivial2 ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. ((∀x∈L.↑P[x])
⇒ (filter(P;L) = L))
S filter_is_nil ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. ((∀x∈L.¬↑P[x])
⇒ (filter(P;L) ≙ []))
S filter_is_empty ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. (↑null(filter(P;L)) ⇐
⇒ ∀i:ℕ||L||. (¬↑(P L[i])))

```
S  filter_is_singleton        ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. ∀x:T.
                                 ((x ∈! L) ⇒ (↑P[x]) ⇒ (∀y∈L.(↑P[y])
                                 ⇒ (y = x)) ⇒ (filter(P;L) = (x::[])))
S  filter_is_singleton2       ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List.
                                 (||filter(P;L)|| = 1 ⇔ ∃i:ℕ||L||.
                                 ((↑(P L[i])) ∧ (∀j:ℕ||L||. ((↑(P L[j]))
                                 ⇒ (i = j)))))
S  filter_sublist             ∀T:𝕌. ∀P:T → 𝔹. ∀L_1,L_2:T List.
                                 (L_1 ⊆ L_2 ⇒ filter(P;L_1) ⊆ filter(P;L_2))
S  filter_is_sublist          ∀T:𝕌. ∀L:T List. ∀P:T → 𝔹. filter(P;L) ⊆ L
S  sublist_filter             ∀T:𝕌. ∀L1,L2:T List. ∀P:T → 𝔹.
                                 (L2 ⊆ filter(P;L1) ⇔ L2 ⊆ L1 ∧
                                 (∀x∈L2.↑(P x)))
S  sublist_filter_set_type    ∀T:𝕌. ∀L1,L2:T List. ∀P:T → 𝔹.
                                 (L2 ⊆ L1 ⇒ (∀x∈L2.↑(P x))
                                 ⇒ L2 ⊆ filter(P;L1))
S  l_before_filter_set_type   ∀T:𝕌. ∀l:T List. ∀P:T → 𝔹. ∀x,y:{x:T| ↑(P x)} .
                                 (x before y ∈ l ⇒ x before y ∈ filter(P;l))
S  sublist*_filter            ∀T:𝕌. ∀P:T → 𝔹. ∀as,bs:T List.
                                 (sublist*(T;as;bs)
                                 ⇒ sublist*(T;filter(P;as);filter(P;bs)))
S  filter_reverse             ∀T:𝕌. ∀L:T List. ∀P:T → 𝔹.
                                 (rev(filter(P;L)) = filter(P;rev(L)))
S  find_wf                    ∀T:𝕌. ∀P:T → 𝔹. ∀as:T List. ∀d:T.
                                 ((first a ∈ as s.t. P[a] else d) ∈ T)
S  find_property              ∀T:𝕌. ∀P:T → 𝔹. ∀as:T List. ∀d:T.
                                 (((first a ∈ as s.t. P[a] else d) ∈ as)
                                 ∨ ((first a ∈ as s.t. P[a] else d) = d))
S  filter2_wf                 ∀T:𝕌. ∀L:T List. ∀P:ℕ||L|| → 𝔹. (filter2(P;L) ∈ T List)
S  cons_filter2               ∀T:𝕌. ∀x:T. ∀L:T List. ∀P:ℕ||L|| + 1
                                   → 𝔹. (filter2(P;x::L)
                                 = if P 0 then x::filter2(λi.(P (i + 1));L)
                                   else filter2(λi.(P (i + 1));L) fi )
S  filter_filter2             ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List.
                                 (filter(P;L) = filter2(λi.(P L[i]);L))
S  member_filter2             ∀T:𝕌. ∀L:T List. ∀P:ℕ||L||
                                 → 𝔹. ∀x:T. ((x ∈ filter2(P;L)) ⇔
                                 ∃i:ℕ||L||. ((x = L[i]) ∧ (↑(P i))))
S  filter2_functionality      ∀A:𝕌. ∀L:A List. ∀f1,f2:ℕ||L||
                                 → 𝔹. ((f1 = f2) ⇒ (filter2(f2;L) = filter2(f1;L)))
S  filter_of_filter2          ∀T:𝕌. ∀L:T List. ∀P:ℕ||L|| → 𝔹.
                                   ∀Q:T → 𝔹.
                                   (filter(Q;filter2(P;L))
                                 = filter2(λi.((P i) ∧_b (Q L[i]));L))

END  filter_thms
```

## 2.14 The directory zip_unzip_thms

```
C  zip_unzip_com
                ==================
                ZIP_UNZIP THEOREMS
                ==================
                All zipping and unzipping theorems.

S  zip_wf                    ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List.
                                       (zip(as;bs) ∈ (T1 × T2) List)
S  zip_length                ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List.
                                       ((||zip(as;bs)|| ≤ ||as||) ∧ (||zip(as;bs)|| ≤ ||bs||))
S  length_zip                ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List. ((||as|| = ||bs||)
                                       ⇒ (||zip(as;bs)|| = ||as||))
S  select_zip                ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List. ∀i:ℕ.
                                       ((i < ||zip(as;bs)||) ⇒ (zip(as;bs)[i] = <as[i], bs[i]>))
S  unzip_wf                  ∀T1,T2:𝕌. ∀as:(T1 × T2) List. (unzip(as) ∈ T1 List × (T2 List))
S  unzip_zip                 ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List.
                                       ((||as|| = ||bs||) ⇒ (unzip(zip(as;bs)) = <as, bs>))
S  zip_unzip                 ∀T1,T2:𝕌. ∀as:(T1 × T2) List.
                                       (zip(unzip(as).1;unzip(as).2) = as)

END  zip_unzip_thms
```

## 2.15 The directory all_exists_thms

```
C  all_exists_com
                ====================
                ALL_EXISTS THEOREMS
                ====================
                This directory contains theorems that either assume a proposition is true for
                all elements in a list and then try to prove a consequent from that fact, or
                assume the existence of an element in the list for which the proposition is true
                and then try to prove a consequent from that fact.

S  list_all_wf              ∀T:𝕌. ∀l:T List. ∀P:T → ℙ. (list_all(x.P[x];l) ∈ ℙ)
S  list_all_iff             ∀T:𝕌. ∀l:T List. ∀P:T → ℙ. (list_all(x.P[x];l)
                                                  ⇔ ∀x:T. ((x ∈ l) ⇒ P[x]))
S  l_all_wf                 ∀T:𝕌. ∀L:T List. ∀P:T → ℙ. ((∀x∈L.P[x]) ∈ ℙ)
S  comb_for_l_all_wf        λT,L,P,z.(∀x∈L.P[x]) ∈ T:𝕌 → L:T List → P:T → ℙ → ↓True →
S  l_all_append             ∀T:𝕌. ∀P:T → ℙ. ∀L1,L2:T List.
                                                  ((∀x∈L1 @ L2.P[x]) ⇔ (∀x∈L1.P[x]) ∧ (∀x∈L2.P[x]))
S  l_all_filter             ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. (∀x∈filter(P;L).↑(P x))
S  l_all_cons               ∀T:𝕌. ∀P:T → ℙ. ∀x:T. ∀L:T List.
                                                  ((∀y∈x::L.P[y]) ⇔ P[x] ∧ (∀y∈L.P[y]))
S  l_all_fwd                ∀T:𝕌. ∀P:T → ℙ. ∀L:T List. ∀x:T.
                                                  ((x ∈ L) ⇒ (∀y∈L.P[y]) ⇒ P[x])
S  l_all_map                ∀A,B:𝕌. ∀f:A → B. ∀L:A List. ∀P:B
                                                  → ℙ. ((∀x∈map(f;L).P[x]) ⇔ (∀x∈L.P[f x]))
S  l_all_nil                ∀T:𝕌. ∀P:T → ℙ. (∀x∈[].P[x])
S  l_all_reduce             ∀T:𝕌. ∀L:T List. ∀P:T → 𝔹.
                                                  ((∀x∈L.↑P[x]) ⇔ ↑reduce(λx,y.(P[x] ∧b y);tt;L))
S  decidable__l_all         A:𝕌. ∀F:A → ℙ. ∀L:A List. ((∀k:A. Dec(F[k])) ⇒ Dec((∀k∈L.F[k])))
S  l_all2_wf                ∀T:𝕌. ∀L:T List. ∀P:T → T → ℙ. ((∀x<y∈L.P[x;y]) ∈ ℙ)
S  l_all2_cons              ∀T:𝕌. ∀L:T List. ∀P:T → T → ℙ. ∀u:T.
                                                  ((∀x<y∈u::L.P[x;y]) ⇔ (∀y∈L.P[u;y]) ∧ (∀x<y∈L.P[x;y]))
S  l_all_since_wf           ∀T:𝕌. ∀P:T → ℙ. ∀L:T List. ∀a:T. ((∀x≥a∈L.P[x]) ∈ ℙ)
S  l_exists_wf              ∀T:𝕌. ∀L:T List. ∀P:T → ℙ. ((∃x∈L.P[x]) ∈ ℙ)
S  l_exists_append          ∀T:𝕌. ∀P:T → ℙ. ∀L1,L2:T List.
                                                  ((∃x∈L1 @ L2.P[x]) ⇔ (∃x∈L1.P[x]) ∨ (∃x∈L2.P[x]))
S  l_exists_nil             ∀T:𝕌. ∀P:T → ℙ. ((∃x∈[].P[x]) ⇔ False)
S  l_exists_cons            ∀T:𝕌. ∀P:T → ℙ. ∀x:T. ∀L:T List.
                                                  ((∃y∈x::L.P[y]) ⇔ P[x] ∨ (∃y∈L.P[y]))
S  l_exists_reduce          ∀T:𝕌. ∀L:T List. ∀P:T → 𝔹.
                                                  ((∃x∈L.↑P[x]) ⇔ ↑reduce(λx,y.(P[x] ∨b y);ff;L))
S  decidable__l_exists      ∀A:𝕌. ∀F:A → ℙ. ∀L:A List. ((∀k:A. Dec(F[k])) ⇒ Dec((∃k∈L.F[k])))

END  all_exists_thms
```

## 2.16 The directory duplicates_thms

```
C  duplicates_com
                ====================
                DUPLICATES THEOREMS
                ====================
                The theorems in this directory all deal with the no_repeats abstraction.

S  no_repeats_wf              ∀T:𝕌. ∀l:T List. (no_repeats(T;l) ∈ ℙ)
S  no_repeats_iff             ∀T:𝕌. ∀l:T List. (no_repeats(T;l) ⇔ ∀x,y:T.
                                          (x before y ∈ l ⇒ (¬(x = y))))
S  no_repeats_cons            ∀T:𝕌. ∀l:T List. ∀x:T. (no_repeats(T;x::l)
                                          ⇔ no_repeats(T;l) ∧ (¬(x ∈ l)))
S  no_repeats_nil             ∀T:𝕌. no_repeats(T;[])
S  no_repeats_append          ∀T:𝕌. ∀l1,l2:T List. (no_repeats(T;l1 @ l2)
                                          ⇒ l_disjoint(T;l1;l2))
S  no_repeats_safety          ∀A:𝕌. safety(A;L.no_repeats(A;L))
S  no_repeats_filter          ∀T:𝕌. ∀P:T → 𝔹. ∀l:T List.
                                          (no_repeats(T;l) ⇒ no_repeats(T;filter(P;l)))
S  no_repeats_single          ∀T:𝕌. ∀x:T. no_repeats(T;x::[])
S  no_reps_cons2              ∀T:𝕌. ∀L:T List. ∀x:T.
                                          (no_repeats(T;x::L) ⇒ (no_repeats(T;L)
                                          ∧ no_repeats(T;x::[])))
S  no_reps_tl                 ∀T:𝕌. ∀L:T List. ∀x:T. (no_repeats(T;x::L) ⇒ no_repeats(T;L))
S  select_equal_no_repeats    ∀T:𝕌. ∀L:T List. ∀i,j:ℕ.
                                          (((i < ||L||) ∧ (j < ||L||)) ⇒ no_repeats(T;L)
                                          ⇒ (L[i] = L[j])
                                          ⇒ (i = j))
S  no_repeats_iseg            ∀T:𝕌. ∀L,L':T List. (L ≤ L' ⇒ no_repeats(T;L') ⇒ no_repeats(T;L))
S  no_reps_append             ∀T:𝕌. ∀L,L':T List.
                                          (no_repeats(T;L @ L')
                                          ⇒ (no_repeats(T;L) ∧ no_repeats(T;L')))
S  no_repeats_append_tl       ∀T:𝕌. ∀L:T List. ∀x:T. (no_repeats(T;L @ (x::[]))
                                          ⇒ no_repeats(T;L))
S  no_repeats_single_append   ∀T:𝕌. ∀L:T List. ∀x:T.
                                          (no_repeats(T;L @ (x::[]))
                                          ⇒ (no_repeats(T;L) ∧ (¬(x ∈ L))))
S  no_reps_reverse            ∀T:𝕌. ∀L:T List. (no_repeats(T;L) ⇒ no_repeats(T;rev(L)))

C  no_repeats_sublist_com
                -------------------
                no_repeats_sublist
                -------------------
                This theorem is interesting as a teaching tool. It is, I believe, unprovable,
                because I made an assumption that a sublist would have no repeated elements.
                However, 'sublist' is only defined in terms of an increasing function, not an
                injective function.

S  no_repeats_sublist         ∀T:𝕌. ∀L,L':T List. (L ⊆ L' ⇒ no_repeats(T;L') ⇒ no_repeats(T;L))
S  no_repeats_member_append   ∀T:𝕌. ∀L,L':T List.
                                          (no_repeats(T;L @ L')
                                          ⇒ ((no_repeats(T;L) ∧ no_repeats(T;L')) ∧
                                          (∀x:T. ((x ∈ L) ⇒ (¬(x ∈ L'))))))

END  duplicates_thms
```

## 2.17 The directory split_thms

```
C  split_com     ===============
                 SPLIT THEOREMS
                 ===============
                 Theorems dealing with the split and split_tail operations.

S  append_split2
                 ∀T:𝕌. ∀L:T List. ∀P:ℕ||L|| → ℙ.
                   ((∀x:ℕ||L||. Dec(P x))
                   ⇒ (∀i,j:ℕ||L||. ((P i) ⇒ (i < j) ⇒ (P j)))
                   ⇒ (∃L_1,L_2:T List
                        ((L = (L_1 @ L_2)) ∧ (∀i:ℕ||L||. (P i ⇔ ||L_1|| ≤ i)))))
S  split_rel_last
                 ∀A:𝕌. ∀r:A → A → 𝔹. ∀L:A List.
                   ((∀a:A. (↑r[a;a]))
                   ⇒ (¬↑null(L))
                   ⇒ (∃L1,L2:A List
                        (((L = (L1 @ L2)) ∧ (¬↑null(L2)) ∧ (∀b∈L2.↑r[b;last(L)]))
                        ∧ ((¬↑null(L1)) ⇒ (¬↑r[last(L1);last(L)])))))
S  append_split ∀T:𝕌. ∀L:T List. ∀P:T → ℙ.
                   ((∀x:ℕ||L||. Dec(P L[x]))
                   ⇒ (∀i,j:ℕ||L||. ((P L[i]) ⇒ (i < j) ⇒ (P L[j])))
                   ⇒ (∃L1,L2:T List
                        (((L = (L1 @ L2)) ∧ (∀i:ℕ||L1||. (¬(P L1[i]))) ∧ (∀i:ℕ||L2||. (P L2[i])))
                        ∧ (∀x∈L.(P x) ⇒ (x ∈ L2)))))
S  split_tail_wf
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List. (split_tail(L | ∀x.f[x]) ∈ A List × (A List))
S  split_tail_trivial
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List.
                   ((∀b:A. ((b ∈ L) ⇒ (↑f[b]))) ⇒ (split_tail(L | ∀x.f[x]) = <[], L>))
S  split_tail_max
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List. ∀a:A.
                   ((a ∈ L)
                   ⇒ (↑f[a])
                   ⇒ (∀b:A. (a before b ∈ L ⇒ (↑f[b])))
                   ⇒ (a ∈ split_tail(L | ∀x.f[x]).2))
S  split_tail_correct
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List. (∀b∈split_tail(L | ∀x.f[x]).2.↑f[b])
S  split_tail_rel
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List.
                   (((split_tail(L | ∀x.f[x]).1) @ (split_tail(L | ∀x.f[x]).2)) = L)
S  split_tail_lemma
                 ∀A:𝕌. ∀f:A → 𝔹. ∀L:A List.
                   (∀a∈L.(∀b≥a∈L.↑f[b])
                   ⇒ (∃L1,L2:A List
                        (((L = (L1 @ L2)) ∧ (a ∈ L2) ∧ (∀b∈L2.↑f[b]))
                        ∧ ((¬↑null(L1)) ⇒ (¬↑f[last(L1)])))))

END  split_thms
```

## 2.18 The directory interleaving_thms

C interleaving_com
```
              ======================
              INTERLEAVING THEOREMS
              ======================
              Theorems dealing with the interleaving abstractions.
```

S interleaving_wf
```
              ∀T:𝕌. ∀L1,L2,L:T List. (interleaving(T;L1;L2;L) ∈ ℙ)
```
S l_before_interleaving
```
              ∀T:𝕌. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L) ⇒ {∀x,y:T. (x before y ∈ L1 ⇒ x before y ∈ L)})
```
S nil_interleaving
```
              ∀T:𝕌. ∀L1,L:T List. (interleaving(T;[];L1;L) ⇔ L = L1)
```
S nil_interleaving2
```
              ∀T:𝕌. ∀L1,L:T List. (interleaving(T;L1;[];L) ⇔ L = L1)
```
S member_interleaving
```
              ∀T:𝕌. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L) ⇒ {∀x:T. ((x ∈ L) ⇔ (x ∈ L1) ∨ (x ∈ L2))})
```
S cons_interleaving
```
              ∀T:𝕌. ∀x:T. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L) ⇒ interleaving(T;x::L1;L2;x::L))
```
S comb_for_interleaving_wf
```
              λT,L1,L2,L,z.interleaving(T;L1;L2;L) ∈ T:
                        → L1:T List → L2:T List → L:T List → ↓True →
```
S length_interleaving
```
              ∀T:𝕌. ∀L,L1,L2:T List. (interleaving(T;L1;L2;L) ⇒ (||L|| = (||L1|| + ||L2||)))
```
S interleaving_of_nil
```
              ∀T:𝕌. ∀L1,L2:T List. (interleaving(T;L1;L2;[]) ⇔ (L1 = []) ∧ (L2 = []))
```
S interleaving_symmetry
```
              ∀T:𝕌. ∀L,L1,L2:T List. (interleaving(T;L1;L2;L) ⇔ interleaving(T;L2;L1;L))
```
S cons_interleaving2
```
              ∀T:𝕌. ∀x:T. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L) ⇒ interleaving(T;L1;x::L2;x::L))
```
S interleaving_of_cons
```
              ∀T:𝕌. ∀x:T. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;x::L)
                    ⇔ ((0 < ||L1||) c∧ ((L1[0] = x) ∧ interleaving(T;tl(L1);L2;L)))
                    ∨ ((0 < ||L2||) c∧ ((L2[0] = x) ∧ interleaving(T;L1;tl(L2);L))))
```
S interleaving_filter2
```
              ∀T:𝕌. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L)
                   ⇔ ∃P:ℕ||L|| → 𝔹. ((L1 = filter2(P;L))
                   ∧ (L2 = filter2(λi.(¬b(P i));L))))
```
S filter_interleaving
```
              ∀T:𝕌. ∀P:T → 𝔹. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L) ⇒ interleaving(T;filter(P;L1);filter(P;L2);filter(P;L)))
```
S interleaving_as_filter
```
              ∀T:𝕌. ∀P:T → 𝔹. ∀L,L1,L2:T List.
                   (interleaving(T;L1;L2;L)
                   ⇒ (∀x∈L2.↑(P x))
                   ⇒ (∀x∈L1.¬↑(P x))
                   ⇒ {(L2 = filter(P;L)) ∧ (L1 = filter(λx.(¬b(P x));L))})
```

```
S  interleaving_as_filter_2
                ∀T:𝕌. ∀P:T → 𝔹. ∀L,L1,L2:T List.
                  (interleaving(T;L1;L2;L)
                  ⇒ (L2 = filter(P;L2))
                  ⇒ (filter(P;L1) = [])
                  ⇒ {(L2 = filter(P;L)) ∧ (L1 = filter(λx.(¬b(P x));L))})
S  sublist_interleaved
                ∀T:𝕌. ∀L,L1:T List. (L1 ⊆ L ⇒ (∃L2:T List. interleaving(T;L1;L2;L)))
S  interleaving_sublist
                ∀T:𝕌. ∀L,L1,L2:T List. (interleaving(T;L1;L2;L) ⇒ L1 ⊆ L)
S  interleaved_split
                ∀T:𝕌. ∀L:T List. ∀P:T → ℙ.
                     ((∀x:T. Dec(P[x]))
                     ⇒ (∃L1,L2:T List
                          (interleaving(T;L1;L2;L)
                          ∧ (∀x:T. ((x ∈ L1) ⇔ (x ∈ L) ∧ P[x]))
                          ∧ (∀x:T. ((x ∈ L2) ⇔ (x ∈ L) ∧ (¬P[x])))))))
S  append_interleaving
                ∀T:𝕌. ∀L1,L2:T List. interleaving(T;L1;L2;L1 @ L2)
S  sublist_append1
                ∀T:𝕌. ∀L1,L2:T List. L1 ⊆ L1 @ L2
S  interleaving_occurence_wf
                ∀T:𝕌. ∀L1,L2,L:T List. ∀f1:ℕ||L1|| → ℕ||L||. ∀f2:ℕ||L2|| → ℕ||L||.
                  (interleaving_occurence(T;L1;L2;L;f1;f2) ∈ ℙ)
S  interleaving_implies_occurence
                ∀T:𝕌. ∀L1,L2,L:T List.
                     (interleaving(T;L1;L2;L)
                     ⇒ (∃f1:ℕ||L1|| → ℕ||L||
                          ∃f2:ℕ||L2|| → ℕ||L||. interleaving_occurence(T;L1;L2;L;f1;f2)))
S  interleaving_occurence_onto
                ∀A:𝕌. ∀L,L1,L2:A List. ∀f1:ℕ||L1|| → ℕ||L||. ∀f2:ℕ||L2|| → ℕ||L||.
                     (interleaving_occurence(A;L1;L2;L;f1;f2)
                     ⇒ (∀j:ℕ||L||. ((∃k:ℕ||L1||. (j = (f1 k))) ∨ (∃k:ℕ||L2||. (j = (f2 k))))))
S  interleaving_split
                ∀T:𝕌. ∀L:T List. ∀P:ℕ||L|| → ℙ.
                     ((∀x:ℕ||L||. Dec(P x))
                     ⇒ (∃L1,L2:T List
                          ∃f1:ℕ||L1|| → ℕ||L||
                          ∃f2:ℕ||L2|| → ℕ||L||
                              (interleaving_occurence(T;L1;L2;L;f1;f2)
                              ∧ ((∀i:ℕ||L1||. (P (f1 i))) ∧ (∀i:ℕ||L2||. (¬(P (f2 i)))))
                              ∧ (∀i:ℕ||L||
                              (((P i) ⇒ (∃j:ℕ||L1||. ((f1 j) = i)))
                              ∧ ((¬(P i)) ⇒ (∃j:ℕ||L2||. ((f2 j) = i))))))))
S  interleaving_singleton
                ∀T:𝕌. ∀L:T List. ∀i:ℕ||L||.
                     ∃L2:T List
                         ∃f1:ℕ1 → ℕ||L||
                         ∃f2:ℕ||L2|| → ℕ||L||
                         (interleaving_occurence(T;L[i]::[];L2;L;f1;f2) ∧ ((f1 0) = i))
```

```
S  last_with_property
               ∀T:𝕌. ∀L:T List. ∀P:ℕ||L|| → ℙ.
                    ((∀x:ℕ||L||. Dec(P x))
                    ⇒ (∃i:ℕ||L||. (P i))
                    ⇒ (∃i:ℕ||L||. ((P i) ∧ (∀j:ℕ||L||. ((i < j) ⇒ (¬(P j)))))))
S  occurence_implies_interleaving
               ∀T:𝕌. ∀L1,L2,L:T List. ∀f1:ℕ||L1|| → ℕ||L||. ∀f2:ℕ||L2|| → ℕ||L||.
                    (interleaving_occurence(T;L1;L2;L;f1;f2) ⇒ interleaving(T;L1;L2;L))
S  filter_is_interleaving
               ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List.
                    interleaving(T;filter(λx.(¬b(P x));L);filter(P;L);L)
S  filter_interleaving_occurence
               ∀T:𝕌. ∀P:T → 𝔹. ∀L:T List.
                    ∃f1:ℕ||filter(λx.(¬b(P x));L)|| → ℕ||L||
                        ∃f2:ℕ||filter(P;L)|| → ℕ||L||
                        (interleaving_occurence(T;filter(λx.(¬b(P x));L);filter(P;L);L;f1;f2)
                        ∧ ((∀i:ℕ||L||
                             ((↑(P L[i]))
                             ⇒ (∃k:ℕ||filter(P;L)||. ((i = (f2 k)) ∧ (L[i] = filter(P;L)[k]))))
                             ∧ (∀i:ℕ||L||
                             ((¬↑(P L[i]))
                             ⇒ (∃k:ℕ||filter(λx.(¬b(P x));L)||
                                  ((i = (f1 k)) ∧ (L[i] = filter(λx.(¬b(P x));L)[k])))))))
                        ∧ (∀i:ℕ||filter(λx.(¬b(P x));L)||. (¬↑(P L[f1 i])))
                        ∧ (∀i:ℕ||filter(P;L)||. (↑(P L[f2 i]))))
S  interleaved_family_occurence_wf
               ∀T,I:𝕌. ∀L:I → (T List). ∀L2:T List. ∀f:i:I → ℕ||L i|| → ℕ||L2||.
                    (interleaved_family_occurence(T;I;L;L2;f) ∈ ℙ)
S  interleaved_family_wf
               ∀T,I:𝕌. ∀L:I → (T List). ∀L2:T List. (interleaved_family(T;I;L;L2) ∈ ℙ)

END  interleaving_thms
```

## 2.19 The directory causal_order_thms

```
C  causal_order_com
                ======================
                CAUSAL ORDER THEOREMS
                ======================
                Theorems dealing with the causal order abstraction.

S  causal_order_wf
                ∀T:𝕌. ∀L:T List. ∀P,Q:ℕ||L|| → ℙ. ∀R:ℕ||L|| → ℕ||L|| → ℙ.
                  (causal_order(L;R;P;Q) ∈ ℙ)
S  causal_order_filter_iseg
                ∀T,T':𝕌. ∀L:T List. ∀P,Q:ℕ||L|| → 𝔹. ∀f,g:T → T'.
                  ((∀L':T List. (L' ≤ L ⇒ map(f;filter2(P;L')) ≤ map(g;filter2(Q;L'))))
                  ⇒ causal_order(L;λi,j.((g L[i]) = (f L[j]));λi.(↑Q[i]);λi.(↑P[i])))
S  causal_order_transitivity
                ∀T:𝕌. ∀L:T List. ∀R:ℕ||L|| → ℕ||L|| → ℙ. ∀P1,P2,P3:ℕ||L|| → ℙ.
                  (Trans(ℕ||L||)(R _1 _2)
                  ⇒ causal_order(L;R;P1;P2)
                  ⇒ causal_order(L;R;P2;P3)
                  ⇒ causal_order(L;R;P1;P3))
S  causal_order_reflexive
                ∀T:𝕌. ∀L:T List. ∀R:ℕ||L|| → ℕ||L|| → ℙ. ∀P:ℕ||L|| → ℙ.
                  (Refl(ℕ||L||)(R _1 _2) ⇒ causal_order(L;R;P;P))
S  causal_order_or
                ∀T:𝕌. ∀L:T List. ∀R:ℕ||L|| → ℕ||L|| → ℙ. ∀P1,P2,P3:ℕ||L|| → ℙ.
                  (Trans(ℕ||L||)(R _1 _2)
                  ⇒ causal_order(L;R;P1;P2)
                  ⇒ causal_order(L;R;P1;P3)
                  ⇒ causal_order(L;R;P1;λi.((P2 i) ∨ (P3 i))))
S  causal_order_sigma
                ∀T,A:𝕌. ∀L:T List. ∀R:ℕ||L|| → ℕ||L|| → ℙ. ∀P,Q:A → ℕ||L|| → ℙ.
                  (Trans(ℕ||L||)(R _1 _2)
                  ⇒ (∀x:A. causal_order(L;R;λi.P[x;i];λi.Q[x;i]))
                  ⇒ causal_order(L;R;λi.∃x:A. P[x;i];λi.∃x:A. Q[x;i]))
S  causal_order_monotonic
                ∀T:𝕌. ∀L:T List. ∀P,Q1,Q2:ℕ||L|| → ℙ. ∀R:ℕ||L|| → ℕ||L|| → ℙ.
                  ((∀i:ℕ||L||. ((Q2 i) ⇒ (Q1 i)))
                  ⇒ causal_order(L;R;P;Q1)
                  ⇒ causal_order(L;R;P;Q2))
S  causal_order_monotonic2
                ∀T:𝕌. ∀L:T List. ∀P1,P2,Q:ℕ||L|| → ℙ. ∀R:ℕ||L|| → ℕ||L|| → ℙ.
                  ((∀i:ℕ||L||. ((P1 i) ⇒ (P2 i)))
                  ⇒ causal_order(L;R;P1;Q)
                  ⇒ causal_order(L;R;P2;Q))
S  causal_order_monotonic3
                ∀T:𝕌. ∀L:T List. ∀P1,P2,Q1,Q2:ℕ||L|| → ℙ. ∀R1,R2:ℕ||L|| → ℕ||L|| → ℙ.
                  ((∀i:ℕ||L||. ((P1 i) ⇒ (P2 i)))
                  ⇒ (∀i:ℕ||L||. ((Q2 i) ⇒ (Q1 i)))
                  ⇒ (∀i,j:ℕ||L||. ((R1 i j) ⇒ (R2 i j)))
                  ⇒ causal_order(L;R1;P1;Q1)
                  ⇒ causal_order(L;R2;P2;Q2))
```

```
S  causal_order_monotonic4
                ∀T:𝕌. ∀L:T List. ∀P1,P2,Q:ℕ||L|| → ℙ. ∀R1,R2:ℕ||L|| → ℕ||L|| → ℙ.
                  ((∀i:ℕ||L||. ((P1 i) ⇒ (P2 i)))
                  ⇒ (∀x,y:ℕ||L||. ((R1 x y) ⇒ (R2 x y)))
                  ⇒ causal_order(L;R1;P1;Q)
                  ⇒ causal_order(L;R2;P2;Q))

END  causal_order_thms
```

## 2.20 The directory permute_swap_thms

C permutations_com
```
                    ====================
                    PERMUTATION THEOREMS
                    ====================
```
Theorems dealing with permutations of lists and swapping the order of elements.

S permute_list_wf — ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. ((L o f) ∈ T List)

S permute_list_select — ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. ∀i:ℕ||L||. ((L o f)[i] = L[f i])

S permute_list_length — ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. (||(L o f)|| = ||L||)

S permute_permute_list — ∀T:𝕌. ∀L:T List. ∀f,g:ℕ||L|| → ℕ||L||. (((L o f) o g) = (L o f o g))

S no_repeats_permute — ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. (Inj(ℕ||L||;ℕ||L||;f) ⇒ no_repeats(T;L) ⇒ no_repeats(T;(L o f)))

C permute_no_repeats_iff_com
```
                    ----------------------
                    no_repeats_permute_iff
                    ----------------------
```
This is an unsolved theorem that has potential to be solved. When I was attempting to solve it, I found I needed a definition of function inverse for one-to-one functions. I could not find one that suited my needs in the library.

S no_repeats_permute_iff — ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. (Inj(ℕ||L||;ℕ||L||;f) ⇒ (no_repeats(T;L) ⇔ no_repeats(T;(L o f))))

S swap_wf — ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. (swap(L;i;j) ∈ T List)

S swap_select — ∀T:𝕌. ∀L:T List. ∀i,j,x:ℕ||L||. (swap(L;i;j)[x] = L[(i, j) x])

S swap_length — ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. (||swap(L;i;j)|| = ||L||)

S swap_swap — ∀T:𝕌. ∀L1:T List. ∀i,j:ℕ||L1||. (swap(swap(L1;i;j);i;j) = L1)

S swapped_select — ∀T:𝕌. ∀L1,L2:T List. ∀i,j:ℕ||L1||. ((L2 = swap(L1;i;j)) ⇒ {(((L2[i] = L1[j]) ∧ (L2[j] = L1[i])) ∧ (||L2|| = ||L1||) ∧ (L1 = swap(L2;i;j))) ∧ (∀x:ℕ||L2||. ((¬(x = i)) ⇒ (¬(x = j)) ⇒ (L2[x] = L1[x])))})

S swap_cons — ∀T:𝕌. ∀L:T List. ∀x:T. ∀i,j:{1..||L|| + 1⁻}. (swap(x::L;i;j) = (x::swap(L;i - 1;j - 1)))

S map_swap — ∀A,B:𝕌. ∀f:B → A. ∀x:B List. ∀i,j:ℕ||x||. (map(f;swap(x;i;j)) = swap(map(f;x);i;j))

S swap_adjacent_decomp — ∀A:𝕌. ∀i:ℕ. ∀L:A List. (((i + 1) < ||L||) ⇒ (∃X,Y:A List ((L = (X @ (L[i]::L[i + 1]::[]) @ Y)) ∧ (swap(L;i;i + 1) = (X @ (L[i + 1]::L[i]::[]) @ Y)))))

S l_before_swap — ∀T:𝕌. ∀L:T List. ∀i:ℕ||L|| - 1. ∀a,b:T. (a before b ∈ swap(L;i;i + 1) ⇒ (a before b ∈ L ∨ ((a = L[i + 1]) ∧ (b = L[i]))))

S comb_for_swap_wf — λT,L,i,j,z.swap(L;i;j) ∈ T: → L:T List → i:ℕ||L|| → j:ℕ||L|| → ↓True → (T List)

```
S  guarded_permutation_wf     ∀T:𝕌. ∀P:L:T List → ℕ||L|| - 1 → ℙ.
                                        (guarded_permutation(T;P) ∈ T List → T List → ℙ)
S  guarded_permutation_transitivity
                              ∀T:𝕌. ∀P:L:T List → ℕ||L|| - 1
                                        → ℙ. Trans(T List)(_1 guarded_permutation(T;P) _2)
S  no_repeats_swap            ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. (no_repeats(T;L)
                                        ⇒ no_repeats(T;swap(L;i;j)))

END  permute_swap_thms
```

## 2.21 The directory index_thms

```
C  index_com     ===============
                 INDEX THEOREMS
                 ===============
                 Theorems dealing with counting the number of elements in a list and finding the
                 index-of-the-first element in a list with a certain property.

S  count_wf                    ∀A:𝕌. ∀P:A → 𝔹. ∀L:A List. (count(P;L) ∈ ℕ)
S  count_index_pairs_wf        ∀T:𝕌. ∀P:L:T List → ℕ||L|| - 1 → ℕ||L||
                                          → 𝔹. ∀L:T List.
                                          (count(i<j<||L|| : P L i j) ∈ ℕ)
S  count_pairs_wf              ∀T:𝕌. ∀L:T List. ∀P:T → T → 𝔹.
                                          (count(x < y in L | P[x;y]) ∈ ℕ)
S  first_index_wf              ∀T:𝕌. ∀L:T List. ∀P:T → 𝔹. (index-of-first x in L.P[x] ∈ ℕ)
S  first_index_cons            ∀T:𝕌. ∀L:T List. ∀a:T. ∀P:T → 𝔹.
                                          (index-of-first x in a::L.P[x] ≜ if P[a] then 1
                                          if 0 <z index-of-first x in L.P[x] then
                                             index-of-first x in L.P[x] + 1
                                          else 0
                                          fi )
S  first_index_equal           ∀T:𝕌. ∀L1,L2:T List. ∀P,Q:T → 𝔹.
                                          ((L1 agree_on(T;a.↑P[a]) L2)
                                          ⇒ (index-of-first a in L1.P[a] ∧b Q[a]
                                           = index-of-first a in L2.P[a] ∧b Q[a]))

END  index_thms
```

## 2.22 The directory length_thms

```
C  stdcomm      ================
                LENGTH THEOREMS
                ================
                All theorems in the standard and list+ libraries pertaining to list length

S  length_wf1                ∀A:𝕌. ∀l:A List. (||l|| ∈ ℤ)
S  comb_for_length_wf1       λA,l,z.||l|| ∈ A:𝕌 → l:A List → ↓True →
S  length_wf2                ||[]|| ∈
S  comb_for_length_wf2       λz.||[]|| ∈ ↓True →
S  length_cons               ∀A:𝕌. ∀a:A. ∀as:A List. (||a::as|| = (||as|| + 1))
S  length_nil                ||[]|| = 0
S  length_of_null_list       ∀A:𝕌. ∀as:A List. ((as = []) ⇒ (||as|| = 0))
S  length_zero               ∀T:𝕌. ∀l:T List. (||l|| = 0 ⇔ l = [])
S  length_of_not_nil         ∀A:𝕌. ∀as:A List. (¬(as = []) ⇔ ||as|| ≥ 1 )
S  non_nil_length            ∀T:𝕌. ∀L:T List. ((¬(L = [])) ⇒ (0 < ||L||))
S  non_neg_length            ∀A:𝕌. ∀l:A List. (||l|| ≥ 0 )
S  pos_length                ∀A:𝕌. ∀l:A List. ((¬(l = [])) ⇒ (||l|| ≥ 1 ))
S  singleton_length          ∀T:𝕌. ∀x:T. (||x::[]|| = 1)
S  l_member_length           ∀T:𝕌. ∀L:T List. ((||L|| ≥ 1 ) ⇒ (∃x:T. (x ∈ L)))
S  length_append             ∀T:𝕌. ∀as,bs:T List. (||as @ bs|| = (||as|| + ||bs||))
S  map_length                ∀A,B:𝕌. ∀f:A → B. ∀as:A List. (||map(f;as)|| = ||as||)
S  map_length_nat            ∀A,B:𝕌. ∀f:A → B. ∀as:A List. (||map(f;as)|| = ||as||)
S  length_tl                 ∀A:𝕌. ∀l:A List. ((||l|| ≥ 1 ) ⇒ (||tl(l)|| = (||l|| - 1)))
S  tl_length2                ∀T:𝕌. ∀L:T List. ((¬(L = [])) ⇒ (||L|| = (||tl(L)|| + 1)))
S  length_nth_tl             ∀A:𝕌. ∀as:A List. ∀n:{0...||as||}. (||nth_tl(n;as)|| = (||as|| - n))
S  length_firstn             ∀A:𝕌. ∀as:A List. ∀n:{0...||as||}. (||firstn(n;as)|| = n)
S  reject_length             ∀T:𝕌. ∀L:T List. ∀i:ℕ. ((0 ≤ i) ⇒ (i < ||L||) ⇒ (¬(L = []))
                                                    ⇒ (||L[i]|| = (||L|| - 1)))
S  length_segment            ∀T:𝕌. ∀as:T List. ∀i:{0...||as||}. ∀j:{i...||as||}.
                                    (||as[i..j⁻]|| = (j - i))
S  iseg_length               ∀T:𝕌. ∀l1,l2:T List. (l1 ≤ l2 ⇒ (||l1|| ≤ ||l2||))
S  iseg_proper_length        ∀T:𝕌. ∀L1,L2:T List. (L1 ≤ L2 ⇒ (||L1|| = ||L2||) ⇒ (L1 = L2))
S  listify_length            ∀T:𝕌. ∀m,n:ℤ. ∀f:{m..n⁻} → T.
                                    ((n < m) ∨ (||listify(f;m;n)|| = (n - m)))
S  list_n_properties         ∀A:𝕌. ∀n:ℤ. ∀as:A List(n). (||as|| = n)
S  mklist_length             ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. (||mklist(n;f)|| = n)
S  length_sublist            ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ (||L1|| ≤ ||L2||))
S  proper_sublist_length     ∀T:𝕌. ∀L1,L2:T List. (L1 ⊆ L2 ⇒ (||L1|| = ||L2||) ⇒ (L1 = L2))
S  length_filter             ∀A:𝕌. ∀P:A → 𝔹. ∀L:A List. (||filter(P;L)|| = count(P;L))
S  zip_length                ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List.
                                    ((||zip(as;bs)|| ≤ ||as||) ∧ (||zip(as;bs)|| ≤ ||bs||))
S  length_zip                ∀T1,T2:𝕌. ∀as:T1 List. ∀bs:T2 List.
                                    ((||as|| = ||bs||) ⇒ (||zip(as;bs)|| = ||as||))
S  length_disjoint_sublists  ∀T:𝕌. ∀L1,L2,L:T List.
                                    (disjoint_sublists(T;L1;L2;L)
                                    ⇒ ((||L1|| + ||L2||) ≤ ||L||))
S  length_interleaving       ∀T:𝕌. ∀L,L1,L2:T List. (interleaving(T;L1;L2;L)
                                    ⇒ (||L|| = (||L1|| + ||L2||)))
S  permute_list_length       ∀T:𝕌. ∀L:T List. ∀f:ℕ||L|| → ℕ||L||. (||(L o f)|| = ||L||)
S  swap_length               ∀T:𝕌. ∀L:T List. ∀i,j:ℕ||L||. (||swap(L;i;j)|| = ||L||)
S  rev_length                ∀T:𝕌. ∀L:T List. (||L|| = ||rev(L)||)
END  length_thms
```

## 2.23 The directory list_dforms

```
C  list_dforms_com
                ==================
                LIST DISPLAY FORMS
                ==================
                This directory contains all of the display forms from both the standard list
                library and the list+ library. Related functions are grouped for easier
                reference, i.e. map_df, mapc_df, and mapcons_df, hd_df and tl_df,
                reduce_df and reduce2_df.
                Note: The select and reject display forms are identical.

D  null_df      EdAlias null ::"<attr-nil>"::  null(<as:as:ALL:!Void():c.(c)>)== null(<as>)
D  cons_df      Prec(L: bd_preop)::Parens ::EdAlias cons ::  {->0}[<h:term>]{<-}== <h>::<t>
                 Prec(L: bd_preop)::Parens ::Hd A::
                   {[SOFT]{->0}[<h:term><#:list:EQUAL:!Void():c.(c)>]{<-}{]}== <h>::<#>
                 Prec(L: bd_preop)::!dform_families(Hd A)::  ; <h:term>== <h>::<t>
                 Prec(L: bd_preop)::!dform_families(Hd A)::
                   ; <h:term><#:list:EQUAL:!Void():c.(c)> == <h>::<#>
                 [{[SOFT]{->0}<head:term> / {NILil:list:EQUAL:!Void():c.(c)>{<-}{]}]== <head>::<tail>
                 [{[SOFT]{->0}<head:term>{<-}{]}] == <head>::[]
                 Hd lisplike::
                   [{[SOFT]{->0}<head:term>; {NILlist:EQUAL:!Void():c.(c)>{<-}{]}]== <head>::<#>
                 !dform_families(Hd lisplike)::
                   {[SOFT]{->0}<head:term> / {NILil:list:EQUAL:!Void():c.(c)>{<-}{]}== <head>::<tail>
                 !dform_not_point_condition::!dform_families(Hd lisplike)::"<attr-nil>"::
                   <head:term>== <head>::[]
                 !dform_families(Hd lisplike)::
                   {[SOFT]{->0}<head:term>; {NILlist:EQUAL:!Void():c.(c)>{<-}{]}== <head>::<#>.
D  list_case    {[SOFT]{->2}case <l:list:ALL:!Void():c.(c)> of {NIL=> {->0}
                 <b:*:ALL:!Void():c.(c)>{<-} {NILh:hd>::<t:tl> => {->0}<r:*:ALL:!Void():c.(c)>
                 {<-} {<-}{NILc{]}
                 == case <l> of [] => <b> | <h>::<t> => <r> esac
D  append_df    Prec(L: inop)::Parens ::
                   {[SOFT]<P:list:LESS:!Void():c.(c)>{NIL{->0}<Q:list:LESS:!Void():c.(c)>{<-}{]}
                   == <P> @ <Q>
                 Hd R::Prec(L: inop)::Parens ::
                   {[SOFT]{->0}<P:list:LESS:!Void():c.(c)>{<-}{NIL<#:list:ALL:!Void():c.(c)>{]}
                   == <P> @ <#>
                 !dform_families(Hd R)::Prec(L: inop)::Parens ::
                   {->0}<P:list:LESS:!Void():c.(c)>{<-}{NIL{->0}<Q:list:LESS:!Void():c.(c)>{<-}
                   == <P> @ <Q>
                 !dform_families(Hd R)::Prec(L: inop)::Parens ::
                   {->0}<P:list:LESS:!Void():c.(c)>{<-}{NIL<#:list:ALL:!Void():c.(c)>
                   == <P> @ <#>
D  length_df    EdAlias length ::"<attr-nil>"::  ||<as:as:ALL:!Void():c.(c)>||== ||<as>||
D  map_df       EdAlias map ::"<attr-nil>"::
                   map(<f:f:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>)== map(<f>;<as>)
D  mapc_df      EdAlias mapc ::"<attr-nil>"::  mapc(<f:f:ALL:!Void():c.(c)>)== mapc(<f>)
D  mapcons_df   EdAlias mapcons ::"<attr-nil>"::
                   mapcons(<f:f:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>)== mapcons(<f>;<as>)
D  hd_df        EdAlias hd ::"<attr-nil>"::  hd(<l:list>)== hd(<l>)
D  tl_df        EdAlias tl ::"<attr-nil>"::  tl(<l:list>)== tl(<l>)
```

```
D  nth_tl_df    EdAlias nth_tl ::"<attr-nil>"::
                   nth_tl(<n:n:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>)== nth_tl(<n>;<as>)
D  reverse_df   EdAlias reverse ::"<attr-nil>"::  rev(<as:as:ALL:!Void():c.(c)>)== rev(<as>)
D  firstn_df    EdAlias firstn ::"<attr-nil>"::
                   firstn(<n:n:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>)== firstn(<n>;<as>)
D  segment_df   EdAlias segment ::Parens ::Prec(L: postop)::
                   <as:as:ALL:!Void():c.(c)>[<m:m:ALL:!Void():c.(c)>..<n:n:ALL:!Void():c.(c)>⁻]
                   == <as>[<m>..<n>⁻]
D  select_df    EdAlias select ::"<attr-nil>"::  <l:list>[<n:nat>]== <l>[<n>]
D  reverse_select_df
                EdAlias reverse_select ::"<attr-nil>"::
                   reverse_select(<l:l:ALL:!Void():c.(c)>;<n:n:ALL:!Void():c.(c)>)
                   == reverse_select(<l>;<n>)
D  reject_df    EdAlias reject ::"<attr-nil>"::
                   <as:as:LESS:!Void():c.(c)>NILi:ALL:!Void():c.(c)>]== <as>[<i>]
D  reject2_df   EdAlias reject2 ::"<attr-nil>"::
                   <bs:bs:LESS:!Void():c.(c)>NILi:ALL:!Void():c.(c)>]== <bs>[<i>]
D  reduce_df    EdAlias reduce ::"<attr-nil>"::
                   reduce(<f:f:ALL:!Void():c.(c)>;<k:k:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>)
                   == reduce(<f>;<k>;<as>)
D  reduce2_df   EdAlias reduce2 ::
                   reduce2(<f:f:ALL:!Void():c.(c)>;<k:k:ALL:!Void():c.(c)>;<i:i:ALL:!Void():c.(c)>;
                   <as:as:ALL:!Void():c.(c)>)
                   == reduce2(<f>;<k>;<i>;<as>)
D  for_df       EdAlias for ::Parens ::Prec(L: bd_preop)::
                   For{<T:T:ALL:!Void():c.(c)>,<op:op:ALL:!Void():c.(c)>,<id:id:ALL:!Void():c.(c)>}
                   <x:var> ∈ <as:T List:ALL:!Void():c.(c)>.{[SOFT]{NIL <f:f:EQUAL:!Void():c.(c)>
                   == For{<T>,<op>,<id>} <x> ∈ <as>. <f>
D  for_hdtl_df EdAlias for_hdtl ::Parens ::Prec(L: bd_preop)::
                   ForHdTl{<A:A:ALL:!Void():c.(c)>,<f:f:ALL:!Void():c.(c)>,<k:k:ALL:!Void():c.(c)>}
                   <h:var>::<t:var> ∈ <as:as:ALL:!Void():c.(c)>.{[SOFT]{NIL <g:g:EQUAL:!Void():c.(c)>
                   == ForHdTl{<A>,<f>,<k>} <h>::<t> ∈ <as>. <g>
D  listify_df   EdAlias listify ::"<attr-nil>"::
                   listify(<f:f:ALL:!Void():c.(c)>;<m:int:ALL:!Void():c.(c)>;<n:int:ALL:!Void():c.(c)>)
                   == listify(<f>;<m>;<n>)
                EdAlias nlistify ::
                   (<f:f:ALL:!Void():c.(c)>)[ℕ<n:nat:ALL:!Void():c.(c)>]== (<f>)[ℕ<n>]
D  list_n_df    EdAlias list_n ::"<attr-nil>"::
                   <A:A:ALL:!Void():c.(c)> List(<n:n:ALL:!Void():c.(c)>)== <A> List(<n>)
D  mklist_df    EdAlias mklist ::
                   mklist(<n:n:ALL:!Void():c.(c)>;<f:f:ALL:!Void():c.(c)>)== mklist(<n>;<f>)
D  l_member_df EdAlias l_member ::"<attr-nil>"::
                   (<x:x:ALL:!Void():c.(c)> ∈ <l:l:ALL:!Void():c.(c)> ∈ <T:T:ALL:!Void():c.(c)>)
                   == (<x> ∈ <l>)
                (<x:x:ALL:!Void():c.(c)> ∈ <l:l:ALL:!Void():c.(c)>)== (<x> ∈ <l>)
                (<x:x:ALL:!Void():c.(c)> foobar <l:l:ALL:!Void():c.(c)>)== ((<x> ∈ <l>) ∈ <l>).
D  l_member!_dfEdAlias l_member! ::
                   l_member!(<x:x:ALL:!Void():c.(c)>;<l:l:ALL:!Void():c.(c)>;<T:T:ALL:!Void():c.(c)>)
                   == (<x> ∈! <l>)
                (<x:x:ALL:!Void():c.(c)> ∈! <l:l:ALL:!Void():c.(c)>)== (<x> ∈! <l>).
```

```
D  agree_on_common_df
                EdAlias agree_on_common ::
                  agree_on_common(<T:T:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>;
                  <bs:bs:ALL:!Void():c.(c)>)
                  == agree_on_common(<T>;<as>;<bs>)
D  agree_on_df EdAlias agree_on ::
                  agree_on(<T:T:ALL:!Void():c.(c)>;<x:var>.<P:P:ALL:!Void():c.(c)>)
                  == agree_on(<T>;<x>.<P>)
D  last_df     EdAlias last ::  last(<L:L:ALL:!Void():c.(c)>)== last(<L>)
D  sublist_df  EdAlias sublist ::
                  <L1:L1:ALL:!Void():c.(c)> ⊆ <L2:L2:ALL:!Void():c.(c)>== <L1> ⊆ <L2>
D  sublist_occurence_df
                EdAlias sublist_occurence ::
                  sublist_occurence(<T:T:ALL:!Void():c.(c)>;<L1:L1:ALL:!Void():c.(c)>;
                  <L2:L2:ALL:!Void():c.(c)>;<f:f:ALL:!Void():c.(c)>)
                  == sublist_occurence(<T>;<L1>;<L2>;<f>)
D  l_subset_df EdAlias l_subset ::
                  l_subset(<T:T:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>;<bs:bs:ALL:!Void():c.(c)>)
                  == l_subset(<T>;<as>;<bs>)
D  sublist*_df EdAlias sublist* ::
                  sublist*(<T:T:ALL:!Void():c.(c)>;<as:as:ALL:!Void():c.(c)>;
                  <bs:bs:ALL:!Void():c.(c)>)
                  == sublist*(<T>;<as>;<bs>)
D  disjoint_sublists_df
                EdAlias disjoint_sublists ::
                  disjoint_sublists(<T:T:ALL:!Void():c.(c)>;<L1:L1:ALL:!Void():c.(c)>;
                  <L2:L2:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                  == disjoint_sublists(<T>;<L1>;<L2>;<L>)
D  l_before_df EdAlias l_before ::"<attr-nil>"::
                  <x:x:ALL:!Void():c.(c)> before <y:y:ALL:!Void():c.(c)> ∈
                  <l:l:ALL:!Void():c.(c)> ∈ <T:T:ALL:!Void():c.(c)>
                  == <x> before <y> ∈ <l>
                <x:x:ALL:!Void():c.(c)> before <y:y:ALL:!Void():c.(c)> ∈ <l:l:ALL:!Void():c.(c)>
                == <x> before <y> ∈ <l>
D  strong_before_df
                EdAlias strong_before ::
                  <x:x:ALL:!Void():c.(c)> << <y:y:ALL:!Void():c.(c)> ∈ <l:l:ALL:!Void():c.(c)>
                  == <x> << <y> ∈ <l>
D  same_order_df
                EdAlias same_order ::
                  same_order(<x1:x1:ALL:!Void():c.(c)>;<y1:y1:ALL:!Void():c.(c)>;
                  <x2:x2:ALL:!Void():c.(c)>;<y2:y2:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>;
                  <T:T:ALL:!Void():c.(c)>)
                  == same_order(<x1>;<y1>;<x2>;<y2>;<L>;<T>)
D  l_succ_df   EdAlias l_succ ::"<attr-nil>"::
                  <y:var>∈<T:T:ALL:!Void():c.(c)> = succ(<x:x:ALL:!Void():c.(c)>) in
                  <l:l:ALL:!Void():c.(c)>{NIL88⇒ <P:P:ALL:!Void():c.(c)>
                  == <y> = succ(<x>) in <l>
                     ⇒ <P>
                <y:var> = succ(<x:x:ALL:!Void():c.(c)>) in <l:l:ALL:!Void():c.(c)>{NIL88⇒
                <P:P:ALL:!Void():c.(c)>
                == <y> = succ(<x>) in <l>
                   ⇒ <P>
```

```
D  listp_df     EdAlias listp ::"<attr-nil>"::  <A:A:ALL:!Void():c.(c)> List+== <A> List
D  count_df     EdAlias count ::
                  count(<P:P:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)== count(<P>;<L>)
D  filter_df    EdAlias filter ::"<attr-nil>"::
                  filter(<P:P:ALL:!Void():c.(c)>;<l:l:ALL:!Void():c.(c)>)== filter(<P>;<l>)
D  filter2_df   EdAlias filter2 ::
                  filter2(<P:P:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)== filter2(<P>;<L>)
D  iseg_df      EdAlias iseg ::"<attr-nil>"::
                  <l1:l1:ALL:!Void():c.(c)> ≤ <l2:l2:ALL:!Void():c.(c)> ∈
                  <T:T:ALL:!Void():c.(c)> List
                  == <l1> ≤ <l2>
                 <l1:l1:ALL:!Void():c.(c)> ≤ <l2:l2:ALL:!Void():c.(c)>== <l1> ≤ <l2>
D  compat_df    EdAlias compat ::
                  <l1:l1:ALL:!Void():c.(c)> || <l2:l2:ALL:!Void():c.(c)>== <l1> || <l2>
D  list_accum_df
                 EdAlias list_accum ::"<attr-nil>"::
                  list_accum(<x:var>,<a:var>.<f:f:ALL:!Void():c.(c)>;<y:y:ALL:!Void():c.(c)>;
                  <l:l:ALL:!Void():c.(c)>)
                  == list_accum(<x>,<a>.<f>;<y>;<l>)
D  zip_df       EdAlias zip ::"<attr-nil>"::
                  zip(<as:as:ALL:!Void():c.(c)>;<bs:bs:ALL:!Void():c.(c)>)== zip(<as>;<bs>)
D  unzip_df     EdAlias unzip ::"<attr-nil>"::  unzip(<as:as:ALL:!Void():c.(c)>)== unzip(<as>)
D  find_df      EdAlias find ::"<attr-nil>"::
                  (first <x:var> ∈ <as:as:ALL:!Void():c.(c)> s.t. <P:P:ALL:!Void():c.(c)>
                   else <d:d:ALL:!Void():c.(c)>)
                  == (first <x> ∈ <as> s.t. <P> else <d>)
D  list_all_df  EdAlias list_all ::"<attr-nil>"::
                  list_all(<x:var>.<P:P:ALL:!Void():c.(c)>;<l:l:ALL:!Void():c.(c)>)
                  == list_all(<x>.<P>;<l>)
D  l_all_df     EdAlias l_all ::
                  l_all(<L:L:ALL:!Void():c.(c)>;<T:T:ALL:!Void():c.(c)>;<x:var>.<P:P:ALL:!Void():c.(c)>)
                  == (∀<x>∈<L>.<P>)
                 EdAlias l_all ::
                  (∀<x:var>∈<L:L:ALL:!Void():c.(c)>.<P:P:ALL:!Void():c.(c)>)== (∀<x>∈<L>.<P>).
D  l_all2_df    EdAlias l_all2 ::
                  l_all2(<L:L:ALL:!Void():c.(c)>;<T:T:ALL:!Void():c.(c)>;<x:var>,<y:var>.
                  <P:P:ALL:!Void():c.(c)>)
                  == (∀<x><<y>∈<L>.<P>)
                 EdAlias l_all2 ::
                  (∀<x:var><<y:var>∈<L:L:ALL:!Void():c.(c)>.<P:P:ALL:!Void():c.(c)>)
                  == (∀<x><<y>∈<L>.<P>).
D  l_all_since_df
                 EdAlias l_all_since ::
                  l_all_since(<L:L:ALL:!Void():c.(c)>;<T:T:ALL:!Void():c.(c)>;
                  <a:a:ALL:!Void():c.(c)>;<x:var>.<P:P:ALL:!Void():c.(c)>)
                  == (∀<x>≥<a>∈<L>.<P>)
                 EdAlias l_all_since ::
                  (∀<x:var>≥<a:a:ALL:!Void():c.(c)>∈<L:L:ALL:!Void():c.(c)>.<P:P:ALL:!Void():c.(c)>)
                  == (∀<x>≥<a>∈<L>.<P>).
D  l_exists_df  EdAlias l_exists ::
                  l_exists(<L:L:ALL:!Void():c.(c)>;<T:T:ALL:!Void():c.(c)>;<x:var>.
                  <P:P:ALL:!Void():c.(c)>)
                  == (∃<x>∈<L>.<P>)
```

```
                EdAlias l_exists ::
                  (∃<x:var>∈<L:L:ALL:!Void():c.(c)>.<P:P:ALL:!Void():c.(c)>)== (∃<x>∈<L>.<P>).
D  no_repeats_df
                EdAlias no_repeats ::
                  no_repeats(<T:T:ALL:!Void():c.(c)>;<l:l:ALL:!Void():c.(c)>)== no_repeats(<T>;<l>)
D  l_disjoint_df
                EdAlias l_disjoint ::
                  l_disjoint(<T:T:ALL:!Void():c.(c)>;<l1:l1:ALL:!Void():c.(c)>;<l2:l2:ALL:!Void():c.(c)>
                  == l_disjoint(<T>;<l1>;<l2>)
D  append_rel_df
                EdAlias append_rel ::
                  append_rel(<T:T:ALL:!Void():c.(c)>;<L1:L1:ALL:!Void():c.(c)>;
                  <L2:L2:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                  == append_rel(<T>;<L1>;<L2>;<L>)
D  safety_df    EdAlias safety ::
                  safety(<A:A:ALL:!Void():c.(c)>;<tr:var>.<P:P:ALL:!Void():c.(c)>)== safety(<A>;<tr>.<P>
D  strong_safety_df
                EdAlias strong_safety ::
                  strong_safety(<T:T:ALL:!Void():c.(c)>;<tr:var>.<P:P:ALL:!Void():c.(c)>)
                  == strong_safety(<T>;<tr>.<P>)
D  mapfilter_dfEdAlias mapfilter ::
                  mapfilter(<f:f:ALL:!Void():c.(c)>;<P:P:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                  == mapfilter(<f>;<P>;<L>)
D  split_tail_df
                EdAlias split_tail ::
                  split_tail(<x:var>.<f:f:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                  == split_tail(<L> | ∀<x>.<f>)
                EdAlias split_tail ::
                  split_tail(<L:L:ALL:!Void():c.(c)> | ∀<x:var>.<f:f:ALL:!Void():c.(c)>)
                  == split_tail(<L> | ∀<x>.<f>).
D  interleaving_df
                EdAlias interleaving ::
                  interleaving(<T:T:ALL:!Void():c.(c)>;<L1:L1:ALL:!Void():c.(c)>;
                  <L2:L2:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                  == interleaving(<T>;<L1>;<L2>;<L>)
D  interleaving_occurence_df
                EdAlias interleaving_occurence ::
                  interleaving_occurence(<T:T:ALL:!Void():c.(c)>;<L1:L1:ALL:!Void():c.(c)>;
                  <L2:L2:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>;<f1:f1:ALL:!Void():c.(c)>;
                  <f2:f2:ALL:!Void():c.(c)>)
                  == interleaving_occurence(<T>;<L1>;<L2>;<L>;<f1>;<f2>)
D  interleaved_family_occurence_df
                EdAlias interleaved_family_occurence ::
                  interleaved_family_occurence(<T:T:ALL:!Void():c.(c)>;<I:I:ALL:!Void():c.(c)>;
                  <L:L:ALL:!Void():c.(c)>;<L2:L2:ALL:!Void():c.(c)>;<f:f:ALL:!Void():c.(c)>)
                  == interleaved_family_occurence(<T>;<I>;<L>;<L2>;<f>)
D  interleaved_family_df
                EdAlias interleaved_family ::
                  interleaved_family(<T:T:ALL:!Void():c.(c)>;<I:I:ALL:!Void():c.(c)>;
                  <L:L:ALL:!Void():c.(c)>;<L2:L2:ALL:!Void():c.(c)>)
                  == interleaved_family(<T>;<I>;<L>;<L2>)
```

```
D  causal_order_df
               EdAlias causal_order ::
                 causal_order(<L:L:ALL:!Void():c.(c)>;<R:R:ALL:!Void():c.(c)>;
                 <P:P:ALL:!Void():c.(c)>;<Q:Q:ALL:!Void():c.(c)>)
                 == causal_order(<L>;<R>;<P>;<Q>)
D  permute_list_df
               EdAlias permute_list ::
                 permute_list(<L:L:ALL:!Void():c.(c)>;<f:f:ALL:!Void():c.(c)>)== (<L> o <f>)
               EdAlias permute_list ::
                 (<L:L:ALL:!Void():c.(c)> o <f:f:ALL:!Void():c.(c)>)== (<L> o <f>).
D  swap_df     EdAlias swap ::
                 swap(<L:L:ALL:!Void():c.(c)>;<i:i:ALL:!Void():c.(c)>;<j:j:ALL:!Void():c.(c)>)
                 == swap(<L>;<i>;<j>)
D  guarded_permutation_df
               EdAlias guarded_permutation ::
                 guarded_permutation(<T:T:ALL:!Void():c.(c)>;<P:P:ALL:!Void():c.(c)>)
                 == guarded_permutation(<T>;<P>)
D  count_index_pairs_df
               EdAlias count_index_pairs ::
                 count_index_pairs(<P:P:ALL:!Void():c.(c)>;<L:L:ALL:!Void():c.(c)>)
                 == count(i<j<||<L>|| : <P> <L> i j)
               EdAlias count_index_pairs ::
                 count(i<j<||<L:L:ALL:!Void():c.(c)>|| : <P:P:ALL:!Void():c.(c)>
                 <L:L:ALL:!Void():c.(c)> i j)
                 == count(i<j<||<L>|| : <P> <L> i j).
D  count_pairs_df
               EdAlias count_pairs ::
                 count_pairs(<L:L:ALL:!Void():c.(c)>;<x:var>,<y:var>.<P:P:ALL:!Void():c.(c)>)
                 == count(<x> < <y> in <L> | <P>)
               EdAlias count_pairs ::
                 count(<x:var> < <y:var> in <L:L:ALL:!Void():c.(c)> | <P:P:ALL:!Void():c.(c)>)
                 == count(<x> < <y> in <L> | <P>).
D  first_index_df
               EdAlias first_index ::
                 first_index(<L:L:ALL:!Void():c.(c)>;<x:var>.<P:P:ALL:!Void():c.(c)>)
                 == index-of-first <x> in <L>.<P>
               EdAlias first_index ::
                 index-of-first <x:var> in <L:L:ALL:!Void():c.(c)>.<P:P:ALL:!Void():c.(c)>
                 == index-of-first <x> in <L>.<P>.

END  list_dforms
```

## 2.24 The directory list_code

```
C  list_code_com
                ==========
                LIST CODE
                ==========
                This directory contains all of the code from both the standard list library and
                the list+ library. Related functions are grouped for easier reference, i.e.
                length_unroll and length_unroll_additions. I have not included a listing of the
                code objects in the library, but they can be found and viewed in the 'list_code'
                directory (not to be confused with the functional library) located inside the
                presentation library.
```

This completes the documentation for the list library.