

Induction, Loop Invariants, and List Sorting

This is a revised version of the October 5 handout #10 that fixes several bugs and fleshes out many missing details.

Induction

We want to be able to prove that an algorithm is *correct* – i.e., that it satisfies the specification of a solution for the problem being solved.

The divide/conquer/glue problem solving strategy naturally leads to recursive algorithms. How does one prove a recursive algorithm is correct?

Induction is a proof methodology for showing that recursive algorithms are correct. An inductive proof has the following structure:

1. Show that the base case(s) of the recursive algorithm are correct.
2. Assuming that the algorithm works correctly for all inputs of size $< n$ (this is called the *inductive hypothesis (IH)*), show that the algorithm is correct for every input of size n .

The *principle of induction* says that (1) and (2) imply that the recursive algorithm is correct for all inputs.

Note that induction formally justifies the “wishful thinking” approach to writing recursive programs that you’ve seen in CS111/CS230.

Induction Example: Factorial

Here is the standard recursive definition of factorial, written in the language Haskell:

```
fact n = if n == 0 then 1 else n * fact(n-1)
```

Specification

To prove that `fact` is correct, we must first have a *formal specification* for what it is supposed to compute. In the case of factorial, we want `fact(n)` to calculate:

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

Note that $0! = \prod_{k=1}^0 k$, which is defined to be 1.

Inductive Proof of the Correctness of fact

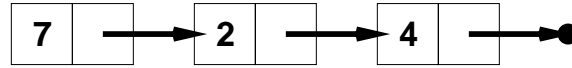
Base Case ($n = 0$): `fact(0)` returns $1 = \prod_{k=1}^0 k = 0!$

Inductive Case ($n > 0$):

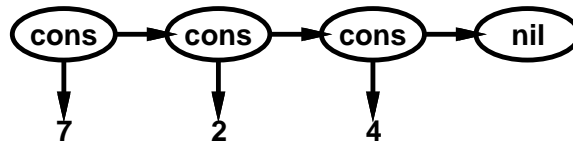
$$\begin{aligned} & n * \text{fact}(n-1) && \text{is returned by } \text{fact}(n) \\ = & n \cdot \prod_{k=1}^{n-1} k && \text{by IH} \\ = & \prod_{k=1}^n k = n! && \text{by algebra} \end{aligned}$$

List Notation

Here is box-and-pointer notation for a list:



Here is a tree (cons/nil) notation for the same list:



Here is Haskell's notation for the same list:

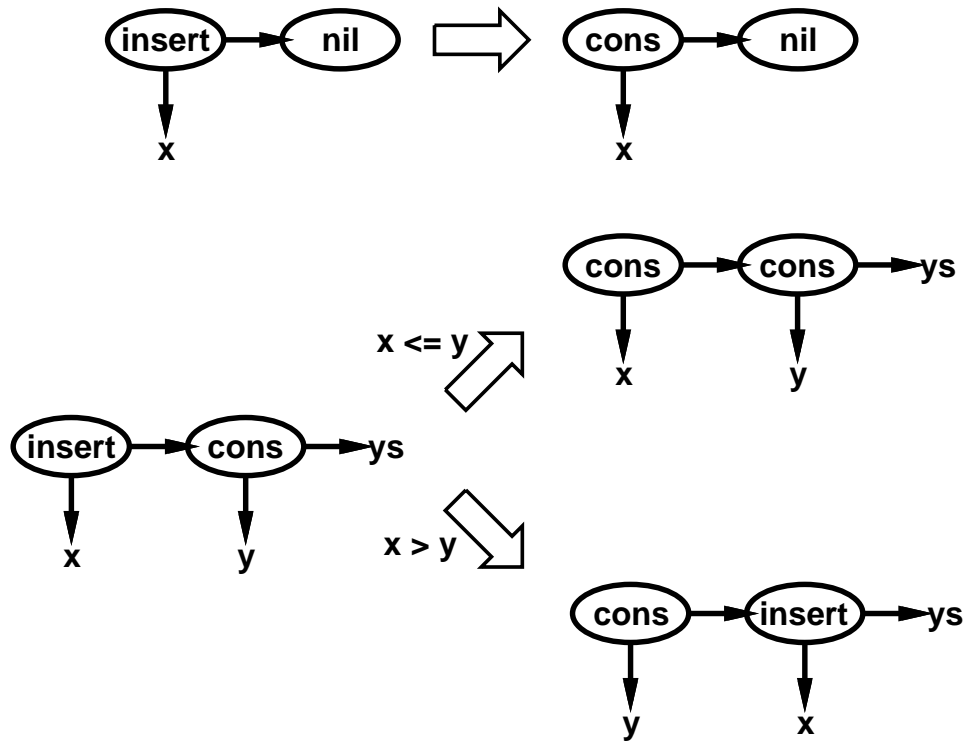
`7 : (2 : (4 : []))`

Here is Haskell's abbreviated notation for the same list:

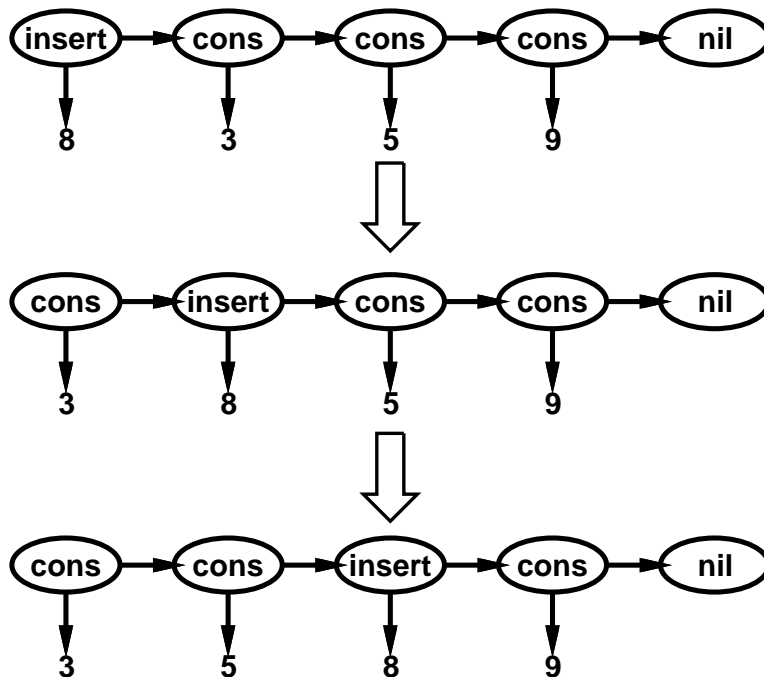
`[7,2,4]`

Graphical List Function Notation

List functions can be written as rewrite rules on the tree notation:



Here's an example using the rules.



Textual List Function Notation

The same insert example from above can be written in Haskell using pattern matching syntax:

```
insert x [] = [x]

insert x (y:ys) =
  if x <= y
  then x:(y:ys)
  else y:(insert x ys)
```

Haskell also allows the second definition clause of `insert` to be written in **guard notation** as

```
insert x (y:ys)
  | x <= y = x:(y:ys)
  | x > y = y:(insert x ys)
```

or as

```
insert x (y:ys)
  | x <= y = x:(y:ys)
  | otherwise = y:(insert x ys)
```

Program Algebra

We can perform algebra on Haskell programs (e.g., substitute equals for equals).

```
insert 8 [3, 5, 9]
⇒ insert 8 (3 : (5 : (9 : [])))
⇒ 3 : (insert 8 (5 : (9 : [])))
⇒ 3 : (5 : (insert 8 (9 : [])))
⇒ 3 : (5 : (8 : (9 : [])))
⇒ [3, 5, 8, 9]
```

This sort of reasoning doesn't work in the presence of side effects (e.g. assigning to a variable, updating a data structure). Unlike Java, C, etc., Haskell does not support side effects.

Sortedness Specification

A list of numbers $xs = [x_1, x_2, \dots, x_k]$ is **sorted**, written $sorted(xs)$, if $x_i \leq x_{i+1}$ for all $i \in [1 .. k - 1]$.

Notes:

- The notation $[x .. y]$ denotes the set of integers from x up to and including y . If $y < x$, then $[x .. y]$ denotes the empty set. E.g.:

$$[3 .. 5] = \{3, 4, 5\}$$

$$[3 .. 3] = \{3\}$$

$$[3 .. 2] = \{\}$$

- The specification for *sorted* does need a special case for empty or singleton lists. Why not?
- Although we have defined sortedness for a list of numbers, everything we say here generalizes to any lists whose elements are “orderable”, but we want to keep things simple here.

Bags (Multisets)

A **bag** (a.k.a. **multiset**) is an unordered collection of elements that may contain multiple occurrence of the same element. In contrast, a set is an unordered collection of elements without duplicates.

We shall use the notation $\{\dots\}$ to denote a bag, where \dots is an enumeration of the elements in the bag. So $\{8, 4, 5, 8, 5, 5\}$ denotes a bag with one 4, three 5s, and two 8s. Since element order doesn't matter, the same bag could be written $\{5, 8, 5, 4, 8, 5\}$ or $\{4, 5, 5, 5, 8, 8\}$.

Bag Operations:

- $bagEmpty$: the empty bag
- $x \in b$: bag membership, true iff there is at least one occurrence of x in b .
- $b_1 =_{bag} b_2$: bag equality, true iff bags b_1 and b_2 have exactly the same elements (including occurrence information).
- $b_1 \cup_{bag} b_2$: the result of unioning all elements of bags b_1 and b_2 .
- $bagIns(x, b)$: the result of inserting one occurrence of element x into bag b .
- $bagDel(x, b)$: the result of removing one occurrence of element x from bag b . (Does nothing if there is no occurrence of x in b .)
- $elts(xs)$: the bag of elements from the list xs . Note that $elts(xs)$ could be written in Haskell syntax as:

```
elts [] = emptyBag
elts (x:xs) = bagIns(x, elts(xs))
```

Specification of insert

Let bs be the result of `insert a as`.

(insert1) If $sorted(as)$ then $sorted(bs)$.

(insert2) $elts(bs) =_{\text{bag}} bagIns(a, elts(as))$.

Correctness Proof for insert: Base Case

Here, $a = x$, $as = []$, and $bs = [x]$.

(insert1) Since $sorted([])$, we must show $sorted([x])$. This is clearly true, since a singleton list is always sorted.

(insert2) $elts([x]) =_{\text{bag}} bagIns(x, elts([]))$. To show this, we can either argue informally that both sides denote a bag with the same single element x , or we can formally justify it by “unwinding” the definition of $elts$.

Correctness Proof for insert: Inductive Case

There are two cases here, depending on whether $x \leq y$ or $x > y$.

Case 1: $x \leq y$.

Here, $a = x$, $as = (y:ys)$, and $bs = x:(y:ys)$.

(insert1) Assuming that $as = (y:ys)$ is sorted, we must show $sorted(x:(y:ys))$. Since $y:ys$ is sorted, we only need show that $x \leq y$. But this is true, by the precondition for Case 1.

(insert2) We must show $elts(x:(y:ys)) =_{\text{bag}} bagIns(x, elts(y:ys))$. It is easy to see this is true, since both sides denote bags containing x , y , and the elements of ys . It can also be shown formally by unwinding the definition of $elts$ on $x:(y:ys)$.

Case 2: $x > y$.

Here, $a = x$, $as = (y:ys)$, and $bs = y:(insert\ x\ ys)$.

(insert1) Assuming that $as = (y:ys)$ is sorted, we must show $sorted(y:(insert\ x\ ys))$. We can show this by proving the following two claims:

1. $sorted(insert\ x\ ys)$: This follows by IH applied to (insert1), which is justified by the fact that ys is smaller than $y:ys$.
2. y is at least as small as any element in $insert\ x\ ys$ (formally: for all z in $elts(insert\ x\ ys)$, $y \leq z$): By IH applied to (insert2), $elts(insert\ x\ ys) =_{\text{bag}} bagIns(x, elts(ys))$. By the precondition of Case 2, $y < x$, and by the assumption that $y:ys$ is sorted, y is \leq any element of ys .

This is an example how the proof of one property – in this case (insert1) – can depend on the induction hypothesis applied to a *different* property – in this case, (insert2).

(insert2) $elts(y:(insert\ x\ ys)) =_{\text{bag}} bagIns(x, elts(y:ys))$. We can reason as follows:

$$\begin{aligned} & elts(y:(insert\ x\ ys)) \\ = & bagIns(y, elts(insert\ x\ ys)) && \text{by the defn. of } elts \\ =_{\text{bag}} & bagIns(y, bagIns(x, elts(ys))) && \text{by IH (since } ys \text{ is smaller than } y:ys \text{) and (insert2).} \\ =_{\text{bag}} & bagIns(x, bagIns(y, elts(ys))) && \text{order of insertion into bag doesn't matter} \\ = & bagIns(x, elts(y:ys)) && \text{by the defn. of } elts \end{aligned}$$

List Sorting Specification

Suppose that `sort(ps)` returns `qs`, where `ps` and `qs` are lists of integers. We say that `sort` is a sorting algorithm iff it satisfies the following two conditions:

(sort1) `sorted(qs)`.

(sort2) `elts(qs) =bag elts(ps)`.

Insertion Sort

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Here is an inductive proof that `isort` satisfies the `sort` specification:

Base Case:

Here, `ps = []` and `qs = []`.

(sort1) `sorted([])` is trivially true.

(sort2) `elts([]) =bag elts([])` is clearly true.

Inductive Case:

Here, `ps = x:xs` and `qs = insert x (isort xs)`.

(sort1) We show `sorted(insert x (isort xs))` via the following steps:

1. `sorted(isort xs)` by IH (sort1) (since `xs` is smaller than `x:xs`).
2. `sorted(isort xs)` implies `sorted(insert x (isort xs))` by (insert1).

(sort2) We show `elts(insert x (isort xs)) =bag elts(x:xs)` as follows:

$$\begin{aligned} & \text{elts}(\text{insert } x \text{ (isort xs)}) \\ =_{\text{bag}} & \text{bagIns}(x, \text{elts}(\text{isort xs})) && \text{by (insert2)} \\ =_{\text{bag}} & \text{bagIns}(x, \text{elts}(\text{xs})) && \text{by IH (isort2) (since xs is smaller than x:xs)} \\ =_{\text{bag}} & \text{elts}(x:\text{xs}) && \text{by defn. of elts} \end{aligned}$$

Quick Sort

```
qsort [] = []
qsort (x:xs) = (qsort ls) ++ [x] ++ (qsort gs)
  where (ls, gs) = partition x xs
```

Notes:

- ++ is Haskell's infix operator for appending two lists. E.g. the result of [7,4,2] ++ [8,4] is [7,4,2,8,4]. You can think of ++ as being defined as follows:

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

- The following fact is useful in correctness proofs of code using ++:

$$elts(xs ++ ys) =_{\text{bag}} elts(xs) \cup_{\text{bag}} elts(ys)$$

- The notation (ls, gs) stands for a *pair* of values. Pattern matching is used to destructure the pair into its component values (here ls and gs).
- The partition function has the following specification.

If (as,bs) = partition p qs, then:

(part1) All elements in as are $\leq p$.

(part2) All elements in bs are $> p$.

(part3) $elts(as) \cup_{\text{bag}} elts(bs) =_{\text{bag}} elts(qs)$

Because p is used to partition the elements of list by magnitude relative to p , it is known as the *pivot*.

Correctness of Quick Sort

Using the specification of `partition`, we prove the correctness of `qsort` by induction.

Base Case: Exactly the same reasoning used in the base case of `isort` applies here.

Inductive Case: Here $ps = x:xs$ and $qs = (\text{qsort } ls) ++ [x] ++ (\text{qsort } gs)$.

(**sort1**) $\text{sorted}((\text{qsort } ls) ++ [x] ++ (\text{qsort } gs))$ follows from these four facts:

1. $\text{sorted}(\text{qsort } ls)$ by IH (sort1)[†].
2. $\text{sorted}(\text{qsort } gs)$ by IH (sort1)[†].
3. All elements of ls (in particular, the last one) are $\leq x$ by (part1).
4. x is $<$ all elements of gs (in particular, the first one) by (part2).

The applications of IH (sort1) marked [†] in steps (1) and (2) require justification. To invoke IH, we must argue that both ls and gs have a length that is strictly less than that of $x:xs$. This follows from (part3): the bag union of ls and gs is xs , so the length of either ls or gs can be at most the length of xs . But the length of xs is strictly less than the length of $x:xs$.

(**sort2**) We show $\text{elts}((\text{qsort } ls) ++ [x] ++ (\text{qsort } gs)) =_{\text{bag}} \text{elts}(x:xs)$ as follows:

$$\begin{aligned} & \text{elts}((\text{qsort } ls) ++ [x] ++ (\text{qsort } gs)) \\ =_{\text{bag}} & \text{elts}(\text{qsort } ls) \cup_{\text{bag}} \text{elts}([x]) \cup_{\text{bag}} \text{elts}(\text{qsort } gs) && \text{by distribution of } \text{elts} \text{ over } ++ \\ =_{\text{bag}} & \text{elts}(ls) \cup_{\text{bag}} \text{elts}([x]) \cup_{\text{bag}} \text{elts}(gs) && \text{by IH (sort2)}^\ddagger \\ =_{\text{bag}} & \text{elts}([x]) \cup_{\text{bag}} \text{elts}(xs) && \text{by (part3)} \\ =_{\text{bag}} & \text{bagIns}(x, \text{elts}(xs)) && \text{by bag algebra} \\ =_{\text{bag}} & \text{elts}(x:xs) && \text{by defn. of } \text{elts} \end{aligned}$$

The applications of IH (sort2) marked [‡] require justification. As in the proof of (sort1), we can argue that both ls and gs have a length that is strictly less than that of $x:xs$.

Partition: Non-Tail-Recursive Version

```
partition1 y [] = ([], [])
partition1 y (z:zs)
  | z <= y = (z:ls, gs)
  | z > y = (ls, z:gs)
  where (ls,gs) = partition1 y zs
```

Example:

```
(ls1,gs1) = partition 5 [7,2,4,6,3]
⇒ (ls2, 7:gs2), where (ls2,gs2) = partition 5 [2,4,6,3]
⇒ (2:ls3, 7:gs3), where (ls3,gs3) = partition 5 [4,6,3]
⇒ (2:(4:ls4), 7:gs4), where (ls4,gs4) = partition 5 [6,3]
⇒ (2:(4:ls5), 7:(6:gs5)), where (ls5,gs5) = partition 5 [3]
⇒ (2:(4:(3:ls6)), 7:(6:gs6)), where (ls6,gs6) = partition 5 []
⇒ (2:(4:(3:[])), 7:(6:[]))
⇒ ([2,4,3], [7,6])
```

Correctness of Non-Tail-Recursive Partition

Base Case:

Here $as = []$, $bs = []$, $p = y$, $qs = []$

- (**part1**) It is trivially true that all elements of $[]$ are $\leq y$
- (**part2**) It is trivially true that all elements of $[]$ are $> y$
- (**part3**) Since $elts([]) = \{\}$, $elts([]) \cup_{\text{bag}} elts([]) =_{\text{bag}} elts([])$

Inductive Case:

From the definition of `partition1`, there are two cases:

Case 1: $z \leq y$

Here $as = z:ls$, $bs = gs$, $p = y$, $qs = z:zs$, where $(ls, gs) = \text{partition1 } y \text{ } zs$.

(**part1**) We show that all elements of $z:ls$ are $\leq y$ as follows:

1. $z \leq y$, by the precondition of Case 1.
2. All elements of ls are $\leq y$, by IH (part 1) (since zs is smaller than $z:zs$).

(**part2**) All elements of gs are $\leq y$ by IH (part 2) (since zs is smaller than $z:zs$).

(**part3**) We show $elts(z:ls) \cup_{\text{bag}} elts(gs) =_{\text{bag}} elts(z:zs)$ as follows:

$$\begin{aligned}
& elts(z:ls) \cup_{\text{bag}} elts(gs) \\
=_{\text{bag}} & \quad bagIns(z, elts(ls)) \cup_{\text{bag}} elts(gs) && \text{by defn. of } elts \\
=_{\text{bag}} & \quad bagIns(z, elts(ls) \cup_{\text{bag}} elts(gs)) && \text{by bag algebra} \\
=_{\text{bag}} & \quad bagIns(z, elts(zs)) && \text{by IH (part3) (since } zs \text{ is smaller than } z:zs) \\
=_{\text{bag}} & \quad elts(z:zs) && \text{by defn. of } elts
\end{aligned}$$

Case 2: $z > y$

Here $as = ls$, $bs = z:gs$, $p = y$, $qs = z:zs$, where $(ls, gs) = \text{partition1 } y \text{ } zs$. The detailed reasoning for this case is similar to that for Case 1 and is omitted.

Partition: Tail-Recursive (Iterative) Version

```
partition2 y ws = partTail y ws [] []
```

```
partTail y [] ls gs = (ls, gs)
```

```
partTail y (z:zs') ls gs
```

```
  | z <= y = partTail y zs' (z:ls) gs
```

```
  | z > y  = partTail y zs' ls (z:gs)
```

Below is an iteration table for `partTail` in the example `partition2 5 [7,2,4,6,3]`. The column labelled `zs` is the list being partitioned.

zs	ls	gs
[7,2,4,6,3]	[]	[]
[2,4,6,3]	[]	[7]
[4,6,3]	[2]	[7]
[6,3]	[4,2]	[7]
[3]	[4,2]	[6,7]
[]	[3,4,2]	[6,7]

How do we prove `partition2` correct? We can use induction, but must be careful – although some arguments decrease in size (`zs`), others increase in size (`ls` and `gs`). Furthermore, certain properties of `ls` and `gs` must be maintained at each recursive call; e.g., the elements of `ls` at each invocation of `partTail` must be $\leq y$.

Specializing induction to the case of loops leads to a proof technique called **loop invariants**.

Loop Invariants

The **loop invariants** proof technique is a specialization of proof-by-induction for iterations (a.k.a loops, a.k.a. tail-recursive functions). Given a loop with state variables s_1, \dots, s_k , the technique involves the following steps. (We use the notation $s_j(i)$ to denote the value of state variable s_j at the beginning of the i th iteration of the loop.)

1. State one or more **invariants** that hold among the state variables of the loop. Formally, this is a k -ary relation $LI(s_1, \dots, s_k)$.
2. Show that the loop invariants hold the first time the loop is entered. I.e., it is necessary to show that $LI(s_1(1), \dots, s_k(1))$ holds.
3. Show that if the loop invariants are true at the beginning of iteration i , then they are true at the beginning iteration $i + 1$, i.e., after the body of the loop has executed, regardless of the path taken through the body. Formally, it must be shown that for all i , $LI(s_1(i), \dots, s_k(i))$ implies $LI(s_1(i + 1), \dots, s_k(i + 1))$.
4. Show that the loop terminates. This is usually done by defining a non-negative integer **metric** function $M(s_1(i), \dots, s_k(i))$. that characterizes the size of the problem at iteration i , and showing that the metric strictly decreases at every iteration.
5. Show that the terminating state satisfies the desired correctness properties. The terminating state corresponds to the beginning of an iteration that is not executed. Formally, if the loop terminates just before the fin th iteration, it must be shown that $LI(s_1(fin), \dots, s_k(fin))$ implies the desired correctness property.

Iterative Factorial

Here is an iterative implementation of the factorial function:

```
fact2 n = factTail n 1

factTail 0 ans = ans
factTail num ans = factTail (num - 1) (num * ans)
```

Below is an iteration table for `factTail` in the invocation `fact2 5`. We assume there is an implicit index variable `i` that counts the iterations of the loop (i.e., the number of calls to `factTail`). We refer to this implicit variable in later discussions.

i	num	ans
1	5	1
2	4	5
3	3	20
4	2	60
5	1	120
6	0	120

Proving partition2 Correct by Loop Invariants

The state variables `num` and `ans` are effectively functions of the implicit index variable i . We write num_i and ans_i for the values of these variables at iteration i .

State invariants

In this example, there is one invariant:

$$\text{(factLI1)} \quad \text{num}_i! \cdot \text{ans}_i = n! \quad (\text{where } n \text{ is the parameter of fact2}).$$

Show invariants hold on entry to loop

$$\text{num}_1! \cdot \text{ans}_1 = n! \cdot 1 = n!.$$

Show each loop iteration preserves invariants

Assume that $\text{num}_i! \cdot \text{ans}_i = n!$. We want to show that $\text{num}_{i+1}! \cdot \text{ans}_{i+1} = n!$. From the definition of `factTail`, we see that $\text{num}_{i+1} = \text{num}_i - 1$ and $\text{ans}_{i+1} = \text{num}_i \cdot \text{ans}_i$. We can combine these facts into the desired proof as follows:

$$\begin{aligned} & \text{num}_{i+1}! \cdot \text{ans}_{i+1} \\ = & (\text{num}_i - 1)! \cdot (\text{num}_i \cdot \text{ans}_i) && \text{by the defn. of factTail} \\ = & \text{num}_i! \cdot \text{ans}_i && \text{by the defn. of factorial} \\ = & n! && \text{by assumption for the } i\text{th iteration} \end{aligned}$$

Show termination:

Define the metric function $M(\text{num}_i, \text{ans}_i) = \text{num}_i$. Since $\text{num}_{i+1} = \text{num}_i - 1$, M clearly decreases with each iteration. Assuming that n is non-negative, `num` is initially non-negative, and, by the definition of `factTail`, `num` never goes below 0. Therefore, `factTail` terminates after a finite number of (n) iterations.

Show desired properties:

When `factTail` terminates in the finth iteration, $\text{num}_t = 0$. So:

$$\begin{aligned} & n! \\ = & \text{num}_{\text{fin}}! \cdot \text{ans}_{\text{fin}} && \text{by (factLI1)} \\ = & 0! \cdot \text{ans}_{\text{fin}} && \text{by base case of factTail} \\ = & \text{ans}_{\text{fin}} && \text{since } 0! = 1 \end{aligned}$$

Therefore, the result returned by `factTail`, the final value of the state variable `ans`, is indeed the factorial of the parameter `n` of `fact2`.

Proving partition2 Correct by Loop Invariants

State invariants There are three invariants that hold among the state variables \mathbf{zs} , \mathbf{ls} , and \mathbf{gs} (each of which is assumed to be indexed by an implicit index variable i).

(partLI1) Every element of \mathbf{ls}_i is \leq pivot y .

(partLI2) Every element of \mathbf{gs}_i is $>$ pivot y .

(partLI3) $elts(\mathbf{ws}) =_{\text{bag}} elts(\mathbf{zs}_i) \cup_{\text{bag}} elts(\mathbf{ls}_i) \cup_{\text{bag}} elts(\mathbf{gs}_i)$, where \mathbf{ws} is the list parameter of `partition2`.

Note how the loop invariants are similar to the specifications (part1), (part2), and (part3). This is not coincidence; each (partLI x) will be used to show (part x) in the final step of the proof.

Show invariants hold on entry to loop In the initial invocation of `partTail`, $\mathbf{zs}_1 = \mathbf{ws}$, $\mathbf{ls}_1 = []$, and $\mathbf{gs}_1 = []$.

(partLI1) Each of the zero elements of $\mathbf{ls}_1 = []$ is $\leq y$.

(partLI2) Each of the zero elements of $\mathbf{gs}_1 = []$ is $>$ pivot y .

(partLI3)

$$\begin{aligned} & elts(\mathbf{ws}) \\ = & elts(\mathbf{ws}) \cup_{\text{bag}} elts([]) \cup_{\text{bag}} elts([]) && \text{by bag algebra} \\ = & elts(\mathbf{zs}_1) \cup_{\text{bag}} elts(\mathbf{ls}_1) \cup_{\text{bag}} elts(\mathbf{gs}_1) && \text{by defn. of partition2} \end{aligned}$$

Show each loop iteration preserves invariants

(partLI1) Assume all elements of \mathbf{ls}_i are $\leq y$. If $\mathbf{z}_i \leq y$, then by the definition of `partTail`, $\mathbf{ls}_{i+1} = \mathbf{z}_i : \mathbf{ls}_i$, and clearly all elements of \mathbf{ls}_{i+1} are $\leq y$. If $\mathbf{z}_i > y$, then by the definition of `partTail`, $\mathbf{ls}_{i+1} = \mathbf{ls}_i$, and the claim holds as well.

(partLI2) Reasoning similar to that used in (partLI1) shows this.

(partLI3) Assume that $elts(\mathbf{ws}) =_{\text{bag}} elts(\mathbf{zs}_i) \cup_{\text{bag}} elts(\mathbf{ls}_i) \cup_{\text{bag}} elts(\mathbf{gs}_i)$. The body of the i th invocation of `partTail` removes the first element \mathbf{z}_i of \mathbf{zs}_i to yield \mathbf{zs}_{i+1} , and either prepends \mathbf{z}_i to \mathbf{ls}_i to yield \mathbf{ls}_{i+1} , or prepends \mathbf{z}_i to \mathbf{gs}_i to yield \mathbf{gs}_{i+1} . In either case, the union of the three bags is unchanged from iteration i to iteration $i + 1$.

Show termination Define the metric function $M(\mathbf{zs}_i, \mathbf{ls}_i, \mathbf{gs}_i) = \text{length}(\mathbf{zs}_i)$. M always returns a non-negative number, and the value strictly decreases with i since the length of \mathbf{zs} decreases by 1 with each iteration. The iteration stops after a finite number of iterations fin when $\text{length}(\mathbf{zs}_{fin}) = 0$.

Show desired properties Assume that the loop terminates right before the fin th iteration, in which case `partition2` returns the pair $(\mathbf{ls}_{fin}, \mathbf{gs}_{fin})$.

- (partLI1) implies that every element of \mathbf{ls}_{fin} is $\leq y$, satisfying (part1).
- (partLI2) implies that every element of \mathbf{gs}_{fin} is $> y$, satisfying (part2).
- (partLI3) implies that $elts(\mathbf{ws}) =_{\text{bag}} elts(\mathbf{zs}_{fin}) \cup_{\text{bag}} elts(\mathbf{ls}_{fin}) \cup_{\text{bag}} elts(\mathbf{gs}_{fin})$. By the base case of `partTail`, $\mathbf{zs}_{fin} = []$. Since $elts([]) =$ the empty bag, bag algebra yields (part3): $elts(\mathbf{ws}) =_{\text{bag}} elts(\mathbf{ls}_{fin}) \cup_{\text{bag}} elts(\mathbf{gs}_{fin})$.

Other Classical Sorting Algorithms

The following are other classical sorting algorithms. The proof of their correctness is left as an exercise.

Selection Sort:

```
ssort [] = []
ssort xs = l:(ssort (del l xs))
  where l = least xs

least [] = error "empty list"
least [y] = y
least (y:ys) = min y (least ys)

del w [] = []
del w (z:zs)
  | w == z    = zs
  | otherwise = z:(del w zs)
```

Merge Sort:

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort zs)
  where (ys,zs) = split xs

merge ps [] = ps
merge [] qs = qs
merge (p:ps) (q:qs)
  | p <= q = p:(merge ps (q:qs))
  | otherwise = q:(merge (p:ps) qs)

split [] = ([], [])
split (x:xs) = (x:zs, ys)
  where (ys, zs) = split xs
```