Lectures Notes on
Inductively-Defined Properties and Verification of Abstract Data Types

Construction and Verification of Software

Nova School of Science and Technology
Mário Pereira mjp.pereira@fct.unl.pt

Version of September 23, 2025

The first goal of this lecture is to learn how to define *logical propositions* over lists and their elements. We pay particular attention to the class of *inductively-defined* propositions.

A second goal is to tackle the verification of functional data structures. The approach we will follow here is commonly referred to as the *algebraic* or *equational* specification approach.

1 Inductive Definitions

So far, we have learned how to write simple, recursive functions that manipulate lists. We have also learned how induction can be used to prove properties about such operations.

During this lecture, we focus our attention on a particular family of functions defined over lists. These capture some logical property about lists and its arguments. In particular, we might be able to *derive a proof* that such property indeed holds. Such kind of functions are also traditionally known as *predicates*.

The type for propositions is a `Rocq` built-in and is referred to using the `Prop` symbol. For instance, if one wants to logically capture what it means for some value to belong to a list, the following recursive function is a possible answer:

Rocq

```
Fixpoint In (a: nat) (l: list nat) : Prop :=
  match l with
  | [ ] ⇒ False
  | b :: m ⇒ b = a ∨ In a m
  end.
```

If we reach the empty list case, then for sure `a` is not in the list. On the other hand, if the head of the list is equal to `a` then we have found it. Otherwise, we must call ourselves recursively for the tail of the initial list. Note that the ∨ actually corresponds to the logical `OR` operator on two propositions. As usual, it states that if its first argument is a valid proposition (in other words, *it holds*), then we do not even need to evaluate the second argument.

At this point, you must be wondering if this is not simply Boolean values. Not at all. Boolean values are simply a program type, which has nothing to do with the notion of truth. While some proof tools actually relax a bit the relation between `bool` and `Prop`, `Rocq` does a very strict distinction between both. The symbol `Bool` is just a type with two constructors, `true` and `false`. Since our first days as programmers we associate the first one with "some claim that must be observed", and the second one with some "some claim we cannot observe". But this is just the semantics that we have chosen to attach to these constructors.

The following table summarizes the key differences between `bool` and `Prop`:

	`bool`	`Prop`
decidable?	yes	no
usable with `match`?	yes	no
equalities rewritable?	no	yes

The most essential difference between the two worlds is decidability. Every `Rocq` expression of type `bool` can be simplified in a finite number of steps to either `true` or `false`, *i.e.*, there is a terminating mechanical procedure for deciding whether or not it is `true`. The second row in the table above follow directly from this essential difference. To evaluate a pattern match (or conditional) on a Boolean, we need to know whether the scrutinee evaluates to `true` or `false`; this only works for `bool`, not `Prop`. The third row highlights another important practical difference: equality functions like `eqb` that return a Boolean cannot be used directly to justify rewriting, whereas the proposition `x = y` can be. This is why we can use `rewrite` when our auxiliary Lemma or hypothesis is an equality.

Lets now look into other examples of propositions defined for lists. The first example is the function that checks whether at least one element of the list respects a predicate `P`. Its recursive definition is as follows:

```
Fixpoint Exists (P: nat → Prop) (l: list nat) : Prop :=          Rocq
  match l with
  | [ ] ⇒ False
  | x :: r ⇒ P x ∨ Exists P r
  end.
```

For instance,

```
Exists (fun e ⇒ e = 0) [1; 1; 0; 1; 0]                           Rocq
```

holds, while

```
Exists (fun e ⇒ e ≥ 10) [1; 2; 3; 4; 5]                          Rocq
```

does not.

Now, even if the above recursive definition is correct, we might want to take a more logical definition of predicate `Exists` and focus only on describing "*how to build a derivation that some element of the list respects some property*". This amounts to use *inductive definitions* or, as we might have heard during Computational Logic, Discrete Mathematics, or Theory of Computation courses, *inductive rules* or *inference rules*. For instance, `Exists` can be inductively defined by the following pair of rules:

$$(\textsc{Exists\_cons\_hd})\ \frac{\mathsf{P}\ x}{\mathtt{Exists}\ \mathsf{P}\ (x :: l)} \qquad \frac{\mathtt{Exists}\ \mathsf{P}\ l}{\mathtt{Exists}\ \mathsf{P}\ (x :: l)}\ (\textsc{Exists\_cons\_tl})$$

This is also a recursive definitions, but these rules immediately induce a proof-tree that `Exists P l` holds for some property `P` and list `l`. The left-hand side rule is the base-case (also known as *axiom*): if the head of the list respects `P`, then we are done; the right-hand side is the recursive case: if we can derive a proof that `Exists P l` holds, then we can conclude that `Exists P (x :: l)` also holds. What about negative cases? That is the big pitfall of inductive definitions: valid derivation trees are only those that are formed using the defined rules and whose branches always terminate with axioms. There is no notion of "tree with holes" or "incomplete trees", hence one cannot derive anything about negative results.

Exercise 1. Using rules (Exists_cons_hd) and (Exists_cons_tl), show that `Exists (fun e => e = 0) [1; 1; 0; 1; 0]` holds. □

Well, the truth is that inductive definitions are pervasive in any mature proof assistant, so in particular our hand-written inductive rules can be easily encoded in Rocq. For the case of `Exists`, this is as follows:

```
Inductive Exists : (nat → Prop) → list nat → Prop :=                Rocq
  | Exists_cons_hd : ∀ P x l, P x → Exists (x :: l)
  | Exists_cons_tl : ∀ P x l, Exists l → Exists (x :: l).
```

Note that `(nat -> Prop) -> list nat -> Prop` means the `Exists` symbol takes a predicate from `nat` to `Prop` as its first argument, a list of natural numbers as its second argument, and finally it returns a proposition. Such definitions are called *inductive predicates*.

Another example of an interesting proposition defined over lists is the `Forall` operator. Once again, this can be defined as a recursive function, as follows:

```
Fixpoint Forall (P: nat → Prop) (l: list nat) : Prop :=             Rocq
  match l with
  | [ ] ⇒ True
  | x :: r ⇒ P x ∧ Forall P r
  end.
```

But once again, we can embrace the logical mantra and encode this an inductive predicate. First, the inductive rules:

$$(\textsc{Forall\_nil})\ \frac{}{\mathtt{Forall}\ \mathsf{P}\ [\,]} \qquad \frac{\mathsf{P}\ l \qquad \mathtt{Forall}\ \mathsf{P}\ l}{\mathtt{Forall}\ \mathsf{P}\ (x :: l)}\ (\textsc{Forall\_cons})$$

and finally the inductive predicate defined in Rocq:

```
Inductive Forall : (nat → Prop) → list nat → Prop :=                Rocq
  | Forall_nil : ∀ P, Forall P [ ]
  | Forall_cons : ∀ P x l,
      P x →
      Forall P l →
      Forall P (x :: l).
```

Exercise 2. Build a derivation that

```
Forall (fun e ⇒ e ≥ 0) [1; 2; 3; 4; 5]
```
Rocq

holds. □

2 Sorted Definition

While it might seem that choosing between recursive or inductive definitions is a matter of style or appealing to some personal preference, the truth is that there some cases where we have no choice other then stick with the inductive approach. Lets try to define what it means, logical, for a list of natural numbers to be sorted in increasing order. Our first attempt might be to write the following `Fixpoint`:

```
Fixpoint sorted (l: list nat) : Prop :=
  match l with
  | [ ] ⇒ True
  | [x] ⇒ True
  | x :: y :: r ⇒ x ≤ y ∧ sorted (y :: r)
  end.
```
Rocq

Unfortunately, this is immediately rejected by the Rocq compiler with the following error:

```
Error:
Recursive definition of sorted is ill-formed.

Recursive call to sorted has principal argument equal to
"y :: r" instead of one of the following variables:
"l0" "r".
```
Rocq

This is our first encounter with the very rigid Rocq termination-checking for recursive functions. Rocq rejects the above definition because it expects the recursion to be done on the sub-term `r` and not on the composed term `(y :: r)`. Without getting into many details, the important message is that every function in Rocq must be *total*, *i.e.*, it always returns some value for all possible inputs. This is exactly the idea behind a mathematical function[1]. For the case of recursive functions, this means that every recursive definitions must be *provably terminating*. Rocq tries to statically check that every `Fixpoint` terminates, simply by inspecting the arguments of recursive calls. But, in order to keep such termination checker decidable, the system is not very smart and rejects some functions that are, indeed, terminating.

So, for the case of `sorted`, we have to define this predicate as an inductive one. The following Rocq definition:

```
Inductive sorted : list nat → Prop :=
  | sorted_nil: sorted [ ]
  | sorted_singleton: ∀ x: nat, sorted [x]
  | sorted_cons: ∀ x y r,
      x ≤ y →
      sorted y :: r →
```
Rocq

[1] If we consider imperative programs, indeed some "functions" might not produce an output or might simply fail for some valid inputs.

sorted $x :: y :: r$.

follows from the following three inductive rules:

$$\frac{}{\texttt{sorted}\ [\,]} \qquad \frac{x : \texttt{nat}}{\texttt{sorted}\ [x]} \qquad \frac{x \leq y \qquad \texttt{sorted}\ (y :: r)}{\texttt{sorted}\ (x :: y :: r)}$$

We can summarize this definition as follows:

- rule (SORTED_NIL) (an axiom) states that the empty list is always sorted.
- rule (SORTED_SINGLETON) (an axiom) states the list `[x]` is always sorted, for any `x` of type `nat`.
- rule (SORTED_CONS) states the list `x :: y :: r` is sorted (for some natural numbers `x` and `y`) if we can show $x \leq y$ and, recursively, that the list `(y :: r)` is also sorted.

Exercise 3. Write a `Fixpoint` version of `sorted`, using the `Forall` predicate. □

Example 2.1. In order to show that list `[1; 2; 3; 4]` is sorted, we build a derivation using the previously defined rules:

$$\dfrac{1 \leq 2 \qquad \dfrac{2 \leq 3 \qquad \dfrac{3 \leq 4 \qquad \dfrac{4 : \texttt{nat}}{\texttt{sorted}\ [4]}\ (\text{SORTED\_SINGLETON})}{\texttt{sorted}\ [3;4]}\ (\text{SORTED\_CONS})}{\texttt{sorted}\ [2;3;4]}\ (\text{SORTED\_CONS})}{\texttt{sorted}\ [1;2;3;4]}\ (\text{SORTED\_CONS})$$

By using inductive rules to define recursive propositions, one actually gets, for free, a very useful *induction principle*. For instance, after the definition of `sorted`, if we instruct Rocq with `Print sorted_ind` we get the following answer:

```
sorted_ind =                                                        Rocq
fun (P : list nat → Prop) (f : P []) (f0 : ∀ x : nat, P [x])
  (f1 : ∀ (x y : nat) (r : list nat),
        x ≤ y → sorted (y :: r) → P (y :: r) → P (x :: y :: r)) ⇒
fix F (l : list nat) (s : sorted l) {struct s} : P l :=
  match s in (sorted l0) return (P l0) with
  | sorted_nil ⇒ f
  | sorted_singleton x ⇒ f0 x
  | sorted_cons x y r l0 s0 ⇒ f1 x y r l0 s0 (F (y :: r) s0)
  end
     : ∀ P : list nat → Prop,
       P [] →
       (∀  x : nat, P [x]) →
       (∀  (x y : nat) (r : list nat),
        x ≤ y → sorted (y :: r) → P (y :: r) → P (x :: y :: r)) →
       ∀ l : list nat, sorted l → P l
```

Do not bother with the details or even try to understand the definition of `sorted_ind`. The important message is that we can perform *induction on the* *derivation* of some logical proposition. In practice, this means the following: suppose we want to prove a Lemma of the form

$$\texttt{sorted}\ l \implies \mathsf{P}$$

for some arbitrary proposition P and some arbitrary list l. Now, if we performs induction on the hypothesis $\mathsf{P}(\texttt{sorted}\ l)$, we get into the following proof state:

- a first case, where $l \equiv [\,]$ and hypothesis `sorted` $[\,]$.
- a second case, where $l \equiv [x]$ and hypothesis `sorted` $[x]$, for some natural number x.
- a third case, where $l \equiv x :: y :: r$ (for some natural numbers x and y, and a list r), the hypothesis `sorted` $(x :: y :: r)$ and, most importantly, the induction hypothesis `sorted` $(y :: r)$.

In fact, induction over inductive definitions should not be surprising at all. A complete derivation using inductive rules is just a recursive object that builds a `proof tree`. Hence, as with any other tree, one might do induction on the structure of the tree or, more precisely in the case of propositions, induction on the *size of the derivation tree*.

3 Functional Heaps

Our main running example during this lecture is a Priority Queue, also know as *Heap*, data structure implemented in a functional way. As our minimal setting, consider the following OCaml interface that establishes the basic building blocks of an heap:

OCaml

```
type heap

val create : heap

val merge : heap -> heap -> heap

val add : nat -> heap -> heap

val remove_min : heap -> heap option
```

What this interface states is that there is a type of `heap` values that can be manipulated using four operations: `create`, `merge`, `add`, and `remove_min`. Lets briefly go trough each of these operations:

- `create` is actually a constant function, without any arguments. For most valid implementations, this function will simply return the empty heap.
- `merge` is the core operation of any heap data structure. It takes two heaps as argument and returns a third queue, which contains a merge of the elements of the argument heaps.
- `add` is a simple operation that inserts a new element in the heap. Since we are under a functional setting, it is worth mentioning this function returns a new heap. In other words, when one calls `add x h`, heap `h` is not lost.

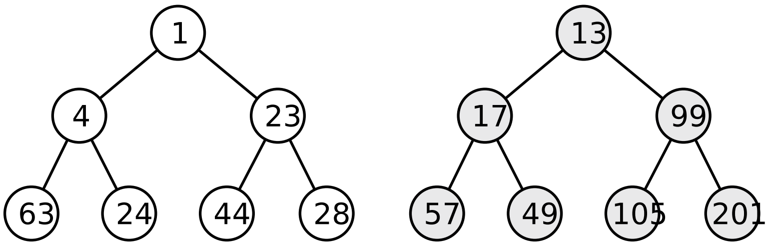

Figure 1: An Example of a Valid Skew Heap.

- finally, the call `remove_min h` returns a new heap without the minimal element of `h`. Note the use of the `option` type for the returned value. This accounts for the case where `h` is the empty heap. Since there is no element to remove, we need to inform a client of this function about it. In an imperative setting we could raise an exception, here we will simply return the `None` value.

In order to illustrate the use and verification of heaps, we will use the *Skew Heaps* variant. This is one of the easiest implementation of heaps, as it amounts to encode heaps using a simple binary tree. The value stored at each node of the tree is then, recursively, less or equal to the value of its descendants. In appendix A, you can find a complete implementation of Skew Heaps in OCaml.

3.1 Skew Heaps Implemented in Rocq

In this section, we present the complete implementation of Skew Heaps in Rocq. We use an auxiliary module `BinTree`, which can be found in appendix B.

Type of heaps. We start with the definition of the heap data type. As mentioned before, a Skew Heap is just a binary tree:

```
Definition heap : Type := BinTree.bin_tree.
```
Rocq

Consider Fig. 1, which presents a schematic example of a Skew Heap. This particular hap can be encoded using our type definition as follows:

```
Definition heap44 : heap :=
  BinTree.Node BinTree.Empty 44 BinTree.Empty.

Definition heap28 : heap :=
  BinTree.Node BinTree.Empty 28 BinTree.Empty.

Definition heap23 : heap :=
  BinTree.Node heap44 23 heap28.

Definition heap1 : heap :=
  BinTree.Node heap4 23 heap23.
```
Rocq

The empty heap. As one might guess, creating a new heap is just a synonym for returning an empty heap. Hence, the Rocq definition of the `create` function is as follows:

```
Definition create : heap := BinTree.Empty.
```
Rocq

Merging two heaps. Merging two heaps is the core operation of any heap implementation. In the case of Skew Heaps, merging is a simple recursive operation. Consider the call `merge h1 h2`. Its output can be described as follows:

- if `h1` is the empty heap, return `h2`;
- if `h2` is the empty heap, return `h2`;
- if `h1` is of the form `Node l1 x1 r1` and `h2` is of the form `Node l2 x2 r2`, then proceed by comparing `x1` with `x2`. If `x1` is less than or equal to `x2`, then build a new heap of the form `Node (merge r1 h2) x1 l1`. Otherwise, build the heap `Node (merge r2 h1) x2 l2`.

Now, if we try to directly translate the OCaml implementation of `merge` into a Rocq fixpoint, the system will complain that it cannot ensure termination. This is due to the fact that in the `then` branch, we do a recursive call on `r1` (a smaller tree than `h1`) and on the `else` branch we do a recursive call on `r2` (a smaller tree than `h2`).

To circumvent this limitation of the Rocq termination-checker, we must resort to encode `merge` as a `Function`. The system will accept our definition, as long as we are able to provide a *termination measure* for all the recursive calls. A termination measure is normally a function that maps the arguments of a `Function` into some natural number. Then, in order to use to show that every recursive call is terminating, the value returned by this measure for the arguments of a recursive should be strictly less than the value returned by the measure for the arguments of initial call.

So, the question now is *what is a good termination measure for the merge function*? Well, whether we follow the `then` branch or the `else` branch, the measure must always decrease. So, we should always take both arguments into account. On the `then` branch, it is the size of `h1` that decreases; on the `else` branch, it is the size of `h2` that decreases. So, one way to always consider both arguments for the measure is to use the sum of their sizes. Here, size means the number of nodes in a binary tree. We provide such measure using the following auxiliary definition:

```
Definition size_two (t: heap * heap) : nat :=
  let (h1, h2) := t in BinTree.size h1 + BinTree.size h2.
```
Rocq

One can now apply it to a `Function` definition of `merge`, as follows:

```
Function merge (t: heap * heap) {measure size_two t} : heap :=
  match t with
  | (BinTree.Empty, h2) => h2
  | (h1, BinTree.Empty) => h1
  | (BinTree.Node l1 x1 r1, BinTree.Node l2 x2 r2) =>
      if x1 <=? x2 then
        BinTree.Node (merge (r1, BinTree.Node l2 x2 r2)) x1 l1
      else
        BinTree.Node (merge (r2, BinTree.Node l1 x1 r1)) x2 l2
  end.
```
Rocq

Immediately after this definition, we enter in proof mode to actually show that all recursive calls indeed halt. There are two cases, one for each recursive call, and the proof proceeds as follows:

```
Proof.                                                    Rocq
  - intros. simpl in *. lia.
  - intros. simpl in *. lia.
Defined.
```

Note that this proof ends up with `Defined`, instead of `Qed`.

Adding a new element to a heap. The two remaining operations in our implementation of Skew Heaps now rely on the use of `merge` as an intermediate routine.

Defining the function that inserts a new element `x` in a heap `h` is as simple as merging `h` with the singleton heap that only contains `x`. The following Rocq definition does it:

```
Definition add (x: nat) (h: heap) : heap :=               Rocq
  merge ((BinTree.Node BinTree.Empty x BinTree.Empty), h).
```

Removing the minimum element of a heap. Finally, the `remove_min` operation should be taken with a grain of salt. What happens if you try to remove the minimum element of an empty heap? Should one raise an exception? Well, out of question since we are using a pure functional language. So, in a functional language, one must use the `option` type to indicate that a function is *partially* defined. Not partial in the sense of `void` from imperative languages, meaning there is no output. Here, partial means that, semantically, it only makes sense to call a function for a subset of its input set of values.

For the case of Skew Heaps, this is as follows:

```
Definition remove_min (h: heap) : option heap :=          Rocq
  match h with
  | BinTree.Empty => None
  | BinTree.Node l _ r => Some (merge (l, r))
  end.
```

In the first branch, where `h` is the empty heap, we return `None` to indicate this function is not defined for the `Empty` argument. In other words, it is a `precondition` that `remove_min` should only be called with non-empty heaps. Finally, for the second branch, we simply return the merge of sub-heaps `l` and `r` wrapped around the `Some` constructor. `Some` is the dual of `None`, which we use to indicate that indeed the function is defined for this given input.

3.2 Logical Specification of Heaps

What it means for some binary tree to be considered a valid Skew Heap? In order words, what distinguishes an arbitrary binary tree from a well-formed heap? To answer these questions, one should revisit the definition of *heap property*: for any given node C, if P is a parent node of C, then the *key* (the value) of P is less than or equal to the key of C. It is worth noting that the previous definition states that *any* parent node P of C must respect the heap property. Indeed, the node at the top of the tree is always the minimum element of the whole data structure. Also, this definition is implicitly a recursive definition that each node key is less than or equal than any of its descendants. The heap presented in Fig. 1 is an example of a valid Skew Heap.

The heap property should be encoded in Rocq either as inductive predicate or as recursive function that returns a value of type `Prop`. Both ways work and are in fact *equivalent*. But, before encoding the heap property, we need to introduce an auxiliary definition that states some natural number is less than or equal to the root of a heap. We can do so using the following inductive rules:

$$(\text{LE\_ROOT\_EMPTY})\ \frac{x : \texttt{nat}}{\texttt{le\_root}\ x\ \texttt{Empty}} \qquad \frac{x \leq y}{\texttt{le\_root}\ x\ (\texttt{Node}\ l\ y\ r)}\ (\text{LE\_ROOT\_NODE})$$

Both rules are axioms. The (LE_ROOT_EMPTY) rule simply states any natural number `x` is smaller than the root of an `Empty` heap, since there is no root at all. The (LE_ROOT_NODE) rule, on the other hand, holds if `x` is less than or equal to the root `y`. This pair of rules can be directly translated into a Rocq inductive definition, as follows:

```
Inductive le_root : nat → heap → Prop :=                                  Rocq
| le_root_empty: ∀ x: nat,
    le_root x BinTree.Empty
| le_root_node : ∀ (x y: nat) (l r: heap),
    x ≤ y →
    le_root x (BinTree.Node l y r).
```

or as the following recursive function:

```
Fixpoint le_root (x: nat) (h: heap) : Prop :=                             Rocq
  match h with
  | BinTree.Empty ⇒ True
  | BinTree.Node l y r ⇒ x ≤ y
  end.
```

Now, we can use the `le_root` to define the heap property for Skew Heaps. We define such property, lets call it `is_heap`, using two inductive rules. First, any `Empty` heap is valid heap:

$$\frac{}{\texttt{is\_heap Empty}}\ (\text{IS\_HEAP\_EMPTY})$$

Second, we need a rule that recursively propagates the heap property for the sub-trees of a non-empty tree. For a heap of the form `Node` l x r, this amounts to state both `l` and `r` are valid heaps (hence, a recursive call to `is_heap`) and that `x` respects the `le_root` property for `l` and `r`. The following rule formally captures this description:

$$\frac{\texttt{is\_heap}\ l \qquad \texttt{is\_heap}\ r \qquad \texttt{le\_root}\ x\ l \qquad \texttt{le\_root}\ x\ r}{\texttt{is\_heap}\ (\texttt{Node}\ l\ x\ r)}\ (\text{IS\_HEAP\_NODE})$$

To encode `is_heap` in Rocq, we can either opt for an inductive definition:

```
Inductive is_heap : heap → Prop :=
| is_heap_empty: is_heap BinTree.Empty
| is_heap_node: ∀ (l: heap) (x: nat) (r: heap),
    le_root x l → le_root x r →
    is_heap l → is_heap r →
    is_heap (BinTree.Node l x r).
```
Rocq

or a recursive function:

```
Fixpoint is_heap (h: heap) : Prop :=
  match h with
  | BinTree.Empty ⇒ True
  | BinTree.Node l x r ⇒
      is_heap l ∧ is_heap r ∧ le_root x l ∧ le_root x r
  end.
```
Rocq

The `is_heap` relation accounts for the *logical specification* of our Skew Heaps implementation. Such predicate forms the backbone of the formal reasoning we shall conduct over operations that manipulate heaps.

3.3 Proof of Correctness for Skew Heaps

Correctness of `create`. The first function we want to prove correct is definitely the easiest one. The correctness of this function simply amounts to state it returns a valid heap, according to the `is_heap` predicate. This is trivial, since the `create` function is just a synonym for the `Empty` constructor. We state and prove such Lemma as follows:

Lemma 3.1 (create_correct)**.** *`is_heap create`.*

Proof.
- We have:

$$\begin{array}{ll} & \texttt{is\_heap correct} \\ \Longrightarrow & \texttt{is\_heap Empty} \quad \text{(by simplification)} \end{array}$$

- By application of rule (IS_HEAP_EMPTY), the following derivation is valid:

$$\frac{}{\texttt{is\_heap Empty}}\ (\textsc{is\_heap\_empty})$$

which finishes the proof.

□

Correctness of `merge`. At this point, we tackle the crucial and most evolved lemma in our implementation. The `merge` operation is a recursive function that takes two heaps as input, lets call them `h1` and `h2`, and should produce as output a third heap, call it `h`. Now, the main point here is that `h` should be a well-formed Skew Heap, in the sense of the `is_heap` relation (this is the correctness conclusion). This is only true if both `h1` and `h2` are also well-formed heaps (the premises for the correction of `merge`). The following Lemma captures exactly the above premises and the conclusion:

Lemma 3.2 (merge_correct). $\forall$ *(*t*: heap * heap) (*h_1 h_2*: heap),*
is_heap $h_1 \implies$ *is_heap* $h_2 \implies$
$t = (h_1, h_2) \implies$
is_heap (merge t*).*

Proof. By functional induction on the call `merge t`.

- Case [t≡`(Empty, h2)`]: immediate, since

  $$\begin{array}{lll} & \texttt{is\_heap (merge t)} & \\ \implies^* & \texttt{is\_heap h2} & \text{(by simplification)} \end{array}$$

  which is directly an hypothesis of the Lemma.

- Case [t≡`(h1, Empty)`]: immediate.

- Case [t≡ `(Node l1 x1 r1, Node l2 x2 r2)`]:

  Here, we proceed by case analysis on the result of `x1` <= ? `x2`. **Note**: in this pen-and-paper style of proofs, we do not distinguish between `bool` and `Prop` values, so `x1 <=? x2 = true` can be treated as x1 ≤ x2 and vice-versa.

  – Sub-case [x1 ≤ x2]:
  We have, by hypothesis:

  $$\begin{array}{rr} \texttt{x1} \leq \texttt{x2} & \text{(Hx1x2)} \\ \texttt{is\_heap (Node l1 x1 r1)} & \text{(Hh1)} \\ \texttt{is\_heap (Node l2 x2 r2)} & \text{(Hh2)} \end{array}$$

  We have, by induction hypothesis:

  $$\begin{array}{l} \forall \texttt{h1}, \texttt{h2} : \texttt{heap} \implies \\ \texttt{is\_heap h1} \implies \\ \texttt{is\_heap h2} \implies \\ \texttt{(r1, Node l2 x2 r2) = (h1, h2)} \implies \\ \texttt{is\_heap (merge (r1, Node l2 x2 r2))} \end{array}$$

  It is worth noting that this induction hypothesis comes directly from the recursive call done in this case.
  We are trying to prove in this case

  ```
  is_heap (Node (merge (r1, Node l2 x2 r2)) x1 l1)
  ```

  Hence, we can apply rule (IS_HEAP_NODE) of predicate `is_heap`. This proceeds as follows:

  $$\dfrac{\begin{array}{cc} \texttt{is\_heap (merge (r1, Node l2 x2 r2))} & \texttt{is\_heap l1} \\ \texttt{le\_root x1 (merge (r1, Node l2 x2 r2))} & \texttt{le\_root x1 l1} \end{array}}{\texttt{is\_heap (Node (merge (r1, Node l2 x2 r2)) x1 l1)}}\ (\text{IS\_HEAP\_NODE})$$

  The first premise, `is_heap (merge (r1, Node l2 x2 r2))`, is proved by application of the induction hypothesis with `h1` instantiated with `r1` and `h2` instantiated with `Node l2 x2 r2`. Proving the premises of the induction hypothesis:

  * `is_heap r1`: by inversion of the hypothesis Hh1, we have directly `is_heap r1`.
  * `is_heap (Node l2 x2 r2)`: by hypothesis Hh2.
  * `(r1, Node l2 x2 r2) = (h1, h2)`: trivial, by the selected instantiation.

The second premise, `is_heap l1`, goes by inversion of the hypothesis Hh1.

The third premise, `le_root x1 (merge (r1, Node l2 x2 r2))`, is proved by case analysis on `r1`:

  * sub-case [`r1`≡`Empty`]:

    `le_root x1 (merge (Empty, Node l2 x2 r2))`
    $\Longrightarrow$ `le_root x1 (Node l2 x2 r2)` (by simpl)

    By application of rule (LE_ROOT_NODE) of predicate `le_root`, we must show `x1` $\leq$ `x2`. This is exactly hypothesis Hx1x2.
  * sub-case [`r1`≡`Node b1 e b2`]:
    We want to prove:

    `le_root x1 (merge (Node b1 e b2, Node l2 x2 r2))`

    Two sub-cases:

    · sub-case [`e` $\leq$ `x2`]:

      `le_root x1 (merge (Node b1 e b2, Node l2 x2 r2))`
      $\Longrightarrow$ `le_root x1 (Node (merge (b2, Node l2 x2 r2)) e b1)`
      (by simplification)

      By application of rule (LE_ROOT_NODE), we must prove `x1` $\leq$ `e`. By doing inversion on hypothesis Hh1, we get exactly `x1` $\leq$ `e`.

    · sub-case [$\neg$`e` $\leq$ `x2`]:

      `le_root x1 (merge (Node b1 e b2, Node l2 x2 r2))`
      $\Longrightarrow$ `le_root x1 (Node (merge (r2, Node b1 e b2)) x2 l2)`
      (by simplification)

      By application of rule (LE_ROOT_NODE), we must prove `x1` $\leq$ `x2`. This is exactly hypothesis Hx1x2.

Finally, the fourth premise, `le_root x1 l1`, goes by inversion on hypothesis Hh1.

– Sub-case [$\neg$`x1` $\leq$ `x2`]: similar to the previous one.

$\square$

3.4 A Digresson on Using Lemmas with Premises

Let us recall the famous *Modus Ponens* rule from Computational Logic

$$\frac{\Gamma \vdash \mathsf{P} \qquad \Gamma \vdash \mathsf{P} \implies \mathsf{Q}}{\Gamma \vdash \mathsf{Q}}$$

where P and Q are some arbitrary logical propositions. In practical terms, this rules tells that if you have a proof that P holds and a proof that P $\implies$ Q holds, then you can immediately conclude that Q holds.

Now, lets do a similar reasoning but backwards. Suppose that you are in the middle of a proof where your goal is to prove that Q holds. Also, you have previously verified a Lemma of the form P $\implies$ Q. What is the missing piece of the puzzle for you to be able to finish your goal? You know that if you are to apply *Modus Ponens*, you finish the proof. So, you just need to derive a proof that P holds and you are good to go.

During a Rocq proof, you will find yourself in many situations like the one described above. For instance, during the proof of the `merge` function, in the Case [`t≡ (Node l1 x1 r1, Node l2 x2 r2)`] and Sub-case [`x1` $\leq$ `x2`], we need to prove the premise that `is_heap (merge (r1, Node l2 x2 r2))`. Your proof state looks something like the following:

```
l1 : BinTree.bin_tree                                              Rocq
x1 : nat
r1, l2 : BinTree.bin_tree
x2 : nat
r2 : BinTree.bin_tree
e0 : (x1 ≤ ? x2) = true
IHh : ∀ h1 h2 : heap,
      is_heap h1 →
      is_heap h2 →
      (r1, BinTree.Node l2 x2 r2) = (h1, h2) → is_heap (merge (r1,
          BinTree.Node l2 x2 r2))
H : is_heap (BinTree.Node l1 x1 r1)
H1 : (BinTree.Node l1 x1 r1, BinTree.Node l2 x2 r2) =
     (BinTree.Node l1 x1 r1, BinTree.Node l2 x2 r2)
H0 : is_heap (BinTree.Node l2 x2 r2)
============================
is_heap (merge (r1, BinTree.Node l2 x2 r2))
```

This is exactly the moment where we want to *apply* the induction hypothesis `IHh`, since the goal matches exactly, up to instantiation, the conclusion of such hypothesis. In Rocq, this is done using the `apply` tactic as follows:

```
apply IHh with (h1 := r1) (h2 := BinTree.Node l2 x2 r2).           Rocq
```

Here, we need to help Rocq a bit in finding the correct instantiation values for parameters `h1` and `h2` of hypothesis `IHh`.

Now, you are left with the following proof state:

```
l1 : BinTree.bin_tree                                                    Rocq
x1 : nat
r1, l2 : BinTree.bin_tree
x2 : nat
r2 : BinTree.bin_tree
e0 : (x1 ≤ ? x2) = true
IHh : ∀ h1 h2 : heap,
      is_heap h1 →
      is_heap h2 →
      (r1, BinTree.Node l2 x2 r2) = (h1, h2) → is_heap (merge (r1,
          BinTree.Node l2 x2 r2))
H : is_heap (BinTree.Node l1 x1 r1)
H1 : (BinTree.Node l1 x1 r1, BinTree.Node l2 x2 r2) =
     (BinTree.Node l1 x1 r1, BinTree.Node l2 x2 r2)
H0 : is_heap (BinTree.Node l2 x2 r2)
============================
is_heap r1

goal 2 (ID 1470) is:
 is_heap (BinTree.Node l2 x2 r2)
goal 3 (ID 1471) is:
 (r1, BinTree.Node l2 x2 r2) = (r1, BinTree.Node l2 x2 r2)
```

where the last two goals follow immediately (the first by using hypothesis `H0`, the second by reflexivity). What the `apply` tactic did was basically apply *Modus Ponens* and made you climb up the proof tree, in order to prove the premise of the `insert_correct` Lemma. This is why we call this style *backwards reasoning*.

Correctness of `add`. After completing the proof of the `merge` function, the correctness of the next two functions follows almost immediately. We begin with proof for the `add` function. This is as follows:

Lemma 3.3 (add_correct)**.** *∀ (x: `nat`) (h: `heap`),*
`is_heap` h ⟹
`is_heap` (`add` x h).

Proof. By hypothesis, we have

$$\texttt{is\_heap h} \tag{H}$$

Now,

```
    is_heap (add x h)
⟹  is_heap (merge ((Node Empty x Empty), h))
```
(by simplification)

The application of lemma `merge_correct`, with `h1` instantiated with `Node Empty x Empty` and `h2` with `h`, finishes the proof. We now have to prove the following premises:

$$\texttt{is\_heap Node Empty x Empty} \tag{Hhx}$$

$$\texttt{is\_heap h} \tag{Hh}$$

The first goes by application of rule (IS_HEAP_NODE) (all the premises of such rule are easily satisfied); the second one is exactly hypothesis H. ☐

Correctness of `remove_min`. The final operation we prove correct, `remove_min`, is the first time we face a branching logical specification. Depending on the output of `remove_min`, it is like we express two different sub-lemmas: if `remove_min` returns `None`, then we know the input heap is the empty heap; otherwise, the input is not the empty heap and the output is of the form `Some h'`. For the latter, `h'` must be a valid Skew Heap.

The following lemma captures this behavior:

Lemma 3.4 (remove_min_correct)**.** ∀ *(*h*: heap),*

```
is_heap h ⟹
match remove_min h with
| None => h = Empty
| Some h′ => is_heap h′
end.
```

Proof. We have, by hypothesis:

$$\texttt{is\_heap h} \tag{H}$$

By case analysis on `h`.

- Case [`h`≡`Empty`]:

  We have

	`match remove_min Empty with ...`	
$\Longrightarrow^*$	`h = Empty`	(by simplification)

  This is exactly the hypothesis of this case.

- Case [`h`≡`Node l x r`]:

  We have

	`match remove_min (Node l x r) with ...`	
$\Longrightarrow^*$	`is_heap (merge (l, r))`	(by simplification)

  By inversion on hypothesis H, we have

  $$\texttt{is\_heap l} \tag{Hl}$$
  $$\texttt{is\_heap r} \tag{Hr}$$
  $$\texttt{le\_root x l} \tag{Hlel}$$
  $$\texttt{le\_root x r} \tag{Hler}$$

  Hence, we can apply lemma `merge_correct` to get

  $$\texttt{is\_heap (merge (l, r))}$$

  where the premises of that Lemma are exactly hypothesis Hl and Hr.

□

A OCaml Implementation of Skew Heaps

```ocaml
type bin_tree =
  | Empty
  | Node of bin_tree * nat * bin_tree

type heap = bin_tree

let create = Empty

let rec merge h1 h2 =
  match h1, h2 with
  | Empty, _ -> h2
  | _, Empty -> h1
  | Node (l1, x1, r1), Node (l2, x2, r2) ->
    if x1 <= x2 then
      Node (merge r1 h2, x1, l1)
    else
      Node (merge r2 h1, x2, l2)

let add x h =
  merge (Node (Empty, x, Empty)) h

let remove_min h =
  match h with
  | Empty -> None
  | Node (l, _, r) -> Some (merge l r)
```

OCaml

B Binary Trees in Rocq

Rocq

```
Module BinTree.

  Inductive bin_tree : Type :=
  | Empty
  | Node (l: bin_tree) (e: nat) (r: bin_tree).

  Fixpoint size (t: bin_tree) : nat :=
    match t with
    | Empty => 0
    | Node l _ r => 1 + size l + size r
    end.

End BinTree.
```