Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

September 23, 2025

Lecture 3

First things first

We have official dates!

Tests:

- Midterm: October 31st (Friday), 6 pm.
- Final test: November 28th (Friday), 6 pm.

Handouts:

- HO1: October 4th (Saturday)
- HO2: November 8th (Saturday)
- HO3: December 4th (Wednesday)

1. Presentation of the first handout

Functional Programming (3/3)

2. Algebraic specification and proof of an ADT
    - inductively defined propositions
    - verified Skew Heaps implementation

We have seen how to define recursive functions to manipulate lists:

```
Fixpoint length (l: list nat) : nat :=
  match l with
  | [ ] ⇒ 0
  | _ :: r ⇒ 1 + length r
  end.

Fixpoint app (l1 l2: list nat) : list nat :=
  match l1 with
  | [ ] ⇒ l2
  | x :: r ⇒ x :: app r l2
  end.
```

and how to make proofs about it:

```
Lemma app_length: ∀(l1 l2: list nat),
  length (app l1 l2) = length l1 + length l2.
Proof. induction l1. ...
```

We can also define propositions on lists:

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

and use such propositions in proofs:

```
Lemma in_cons: ∀(a b: nat) (l: list nat),
  In b l → In b (a :: l).
Proof. simpl. auto. Qed.
```

A common higher-order function on lists:

```
Fixpoint map {A B: Type} (f: A → B) (l: list A) : list B :=
  match l with
  | [] ⇒ []
  | x :: xs ⇒ f x :: map f xs
  end.
```

A property about such function:

```
Lemma in_map_iff: ∀[A B : Type] (f : A → B) (l : list A) (y : B),
  In y (map f l) ↔ (∃ x : A, f x = y ∧ In x l)
```

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

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

Let's take a more logical or proof-oriented approach:

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

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

Here, we focus on the derivation of a proof that `Exists P l` holds.

$$\frac{\text{(Forall\_nil)}}{\texttt{Forall P [ ]}} \qquad \frac{\text{(Forall\_cons)} \quad \texttt{P x} \qquad \texttt{Forall P l}}{\texttt{Exists P (x :: l)}}$$

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

But is not always a matter of style!

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

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

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

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

Verification of Abstract Data Types

ADT: Abstract Data Types

Main point: abstract barrier

- interface
- modularity
- reusability

The interface

```
type heap

val create : heap

val merge : heap -> heap -> heap

val add : nat -> heap -> heap

val remove_min : heap -> heap option
```

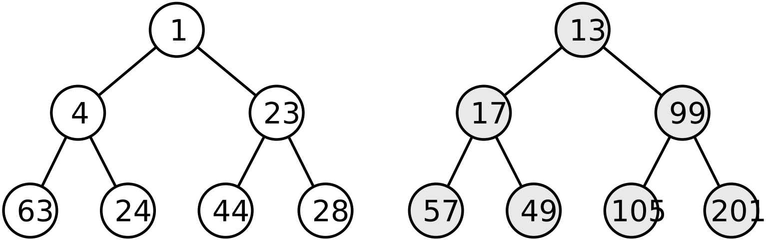
The interface + a specification

```
type heap
(* The data type definition *)

val create : heap
(* [create] returns a new, empty heap *)

val merge : heap -> heap -> heap
(* [merge h1 h2] takes two valid heaps [h1] and [h2]
   and returns a valid heap, which is the result
   of merging the elements of [h1] and [h2] *)

val add : nat -> heap -> heap
(* [add x h] creates a new valid heap, which
   results from inserting [x] into the valid heap [h] *)

val remove_min : heap -> heap option
(* [remove_min h] removes the minimum element of [h],
   if this is not the empty heap; otherwise returns [None] *)
```

Possible implementation for the given interface: Skew Heaps.

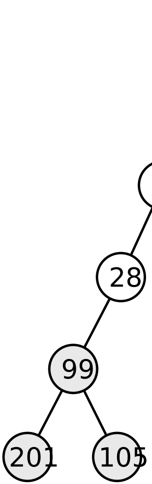
Skew Heaps encode heaps using simple binary trees.

Example:



The most important operation in any heap implementation: merge.

Example (after merge):



Correctness Criterion

What does it mean for a heap implementation to be correct?

The 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.

Every input heap should respect the heap property.

Every output heap should respect the heap property.

Auxiliary definition `le_root`:

$$\frac{x : \texttt{nat}}{\texttt{le\_root}\ x\ \texttt{Empty}}\ (\textsc{le\_root\_empty})$$

$$\frac{x \leq y}{\texttt{le\_root}\ x\ (\texttt{Node}\ l\ y\ r)}\ (\textsc{le\_root\_node})$$

The main definition `is_heap`:

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

$$\frac{\texttt{is\_heap}\ l \qquad \texttt{is\_heap}\ r \qquad \texttt{le\_root}\ x\ l \qquad \texttt{le\_root}\ x\ r}{\texttt{is\_heap}\ (\texttt{Node}\ l\ x\ r)}\ (\textsc{is\_heap\_node})$$

Bibliography

- Software Foundations, Volume 1:
  `https://softwarefoundations.cis.upenn.edu/lf-current/index.html`
  (Chapters *Logic in Coq* and *Inductively Defined Propositions*)

- Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1999.