Handout 1
Verified Proper Cuts

Software Verification

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

Version of September 21, 2025

The goal of this handout is to verify a Rocq implementation of the Proper Cuts problem. Proper Cuts is a classic functional programming challenge: given a list `l`, produce a list `r` of pairs of lists such that, for every element `(l_1, l_2)` in `l`, we have `l_1 ++ l_2 = l`. For instance, consider list `l = [x; y; w]` (the type of the list elements do not really matter here). The list of Proper Cuts for `l` is

```
[([], [x; y; w]); ([x], [y; w]); ([x; y], [w]); ([x; y; w], [])]
```

For reference, we provide a complete OCaml implementation of Proper Cuts in Appendix A. It is strongly advised that we study such an implementation and try it on your own to fully understand the challenge.

During this handout you are supposed to complete definitions and proofs, given in the companion `.v` file. The Proper Cuts implementation is already given. **Do not change such implementation**.

Each exercise in this handout poses a challenge to be addressed in the given Rocq file: either a definition that you must complete, or a proof that you must finish. In the `.v` file, you will find several times the comment

```
(* FILL HERE *)
```

with the corresponding exercise number. This is exactly where you are supposed to write your solution.

General note on polymorphic arguments: you may notice that throughout the companion `.v` file and in this document, many definitions and lemmas annotated with extra arguments of the form `A: Type`. This stands for the use of *polymorphic arguments* in Rocq. In fact, our Proper Cuts implementation is completely agnostic to the type of elements in the list, so does our definitions and lemmas. You can safely ignore such annotations, as all the proofs and definitions you are supposed to provide do not rely on any kind of type information. If you curious, you can read more about Rocq polymorphism in the First Volume of the Software Foundations textbook *Logical Foundations*, chapter *Polymorphism and Higher-Order Functions*[1].

[1] `https://softwarefoundations.cis.upenn.edu/lf-current/Poly.html`

1 Injective Property

The injectivity property states that for a given function `f`, whose domain are values of type `A`, for every two elements `x` and `x'` of type A such that `x` $\neq$`x'`, then their images through `f` are also different. This is exactly the injectivity property that you might have learned in high school.

The Rocq definition of injectivity is already given and is stated as follows:

```
Definition injective [A B: Type] (f: A → B) : Prop :=
  ∀ x x': A, x ≠x' → f x ≠f x'.
```
Rocq

Do not change it. Use it as you find suitable. In our particular case, this property is very useful to show that our implementation of Proper Cuts never produces duplicated elements in the cuts list.

2 Distinct Property

The `distinct` property states that *"the elements of a given list are* pair-wise *distinct"*. In other words, the list does not contain duplicated elements. Mathematically, such a property can be defined using the following three rules"

$$\textsc{Distinct\_nil}\ \frac{}{\texttt{distinct}\ [\,]} \qquad \frac{}{\texttt{distinct}\ [\texttt{x}]}\ \textsc{Distinct\_singleton}$$

$$\textsc{Distinct\_cons}\ \frac{\neg(\texttt{In}\ \texttt{x}\ \texttt{l}) \qquad \texttt{distinct}\ \texttt{l}}{\texttt{distinct}\ (\texttt{x} :: \texttt{l})}$$

The above rules stand for three different cases:

1. an empty list is always `distinct`
2. a list containing only one element is always `distinct`
3. a list of the form `x::l` is `distinct` if `x` is not in `l` and if `l` is `distinct`.

In rule [Distinct_cons], the `In` is the function that decides whether a value is in a list, as seen in lectures. Finally, the $\neg$ symbol stands for the *logical negation* of some property. Its Rocqequivalent is the tilde, $\sim$, symbol.

Exercise 1. Complete the following Rocqdefinition

```
Inductive distinct {A: Type}: list A → Prop := .
```
Rocq

according to rules [Distinct_nil], [Distinct_singleton], and [Distinct_cons]. □

3 Logical Specification

In this handout, we will use several auxiliary definitions to capture the logical properties that a Proper Cuts implementation should observe. First, we define the property *"what is a cut"*, as follows:

```
Definition cut [A: Type] (l1 l2: list A) (l: list A) : Prop :=
  l1 ++l2 = l.
```
Rocq

Second, we introduce an *alias type* definition for a *list of cuts*, to ease our development. This is as follows:

```
Definition cut_list (A: Type) : Type := list (list A * list A).
```
Rocq

Finally, we define the main property of the Proper Cuts algorithm. It states that, for a given cut list `c` and a list `l`

1. `c` is a `distinct` list
2. for every element `(l1, l2)` of `c`, then the pair `(l1, l2)` is a `cut` of l.

This is defined in Rocqas follows:

```
Definition proper_cuts_prop {A: Type}
  (c: cut_list A) (l: list A) : Prop :=
  distinct c ∧ ∀ (l1 l2: list A), In (l1, l2) c → cut l1 l2 l.
```
Rocq

The above definitions are already provided in the companion `.v` file. **Do not change them**.

4 Verified Proper Cuts Operations

4.1 Proper Cuts Implementation, in Rocq

Following the OCaml reference implementation in Appendix A, we first introduce a `cons` function as follows:

```
Definition cons {A: Type}
  (x: A) (l: (list A * list A)) : (list A * list A) :=
  let (l1, l2) := l in (x :: l1, l2).
```
Rocq

This function is useful when composed with the `map` operation, as we shall see later. In practice, the `cons` function is used to inject a value `x` as the head of a list `l1`, where `l1` is the first element of a pair `(l1, l2)`.

Finally, for a given list `l`, the main `proper_cuts` function works as follows:

- if `l` is the empty list, then its Proper Cuts is the list `[([], [])]`, *i.e.*, it only contains the pair `([], [])`.
- if `l` is of the form `x::r`, then
  1. let `pr` be the Proper Cuts list for the tail `r`
  2. let `r` be the list where `x` has been injected into `pr`, using the partial application `(cons x)`
  3. the final Proper Cuts list of `l` is `([], l):: r`

The `proper_cuts` function is defined in Rocqas follows:

```
Fixpoint proper_cuts {A: Type} (l: list A) : cut_list A :=
  match l with
  | [] ⇒ [([], [])]
  | x :: xs ⇒
    let pr := proper_cuts xs in
    let r := map (cons x) pr in
    ([], l) :: r
  end.
```
Rocq

The above definitions are already provided in the companion `.v` file. **Do not change them**.

4.2 Injectivity of `cons`

Consider the expression `(cons x)`. This is a partial application of `cons`, hence it returns a function that expects a *cut pair* as its argument. If two distinct pairs are given as the input of function `(cons x)`, then the outputs must be different. In other words, `(cons x)` *is injective*.

Exercise 2. Prove the following Lemma about the `cons` function:

```
Lemma injective_cons : ∀ {A: Type} (x: A),
  injective (cons x).
```
Rocq

□

4.3 Correctness of `cons`

If the second argument of the `cons` function is a cut pair `(l1, l2)` for some list `l`, then the call `cons x (l1, l2)` must return a pair `(l1', l2')` such that `(l1', l2')` is a cut pair of list `x::l`.

Exercise 3. Prove the following Lemma about the `cons` function:

```
Lemma proper_cuts_cons : ∀ {A: Type}
  (x: A) (l1 l2 l1' l2': list A) (l: list A),
  cut l1 l2 l →
  cons x (l1, l2) = (l1', l2') →
  cut l1' l2' (x::l).
```
Rocq

□

4.4 Composition of the `cons` and `map` functions

When using `(cons x)` as the argument of function `map`, we want it to produce a new list of Proper Cuts, by injecting `x` into the first element of every cut pair.

Exercise 4. Prove the following Lemma on the use of `cons` and `map` functions:

```
Lemma map_cons : ∀ {A: Type}
  (x: A) (l1 l2: list A) (l: cut_list A),
  In (l1, l2) (map (cons x) l) ↔
  (∃ s1: list A, l1 = x :: s1 ∧ In (s1, l2) l).
```
Rocq

□

4.5 Correctness of `proper_cuts`

Finally, the `proper_cuts` function must respect the `proper_cuts_prop` we have previously defined. This entails the correctness of the Proper Cuts algorithm.

Exercise 5. Prove the following Lemma about the `proper_cuts` function:

```
Lemma proper_cuts_correct : ∀ {A: Type} (l: list A),
  proper_cuts_prop (proper_cuts l) l.
```
Rocq

□

4.6 Bonus Exercise

In the companion `.v` file, one can find the `injective_map` property that states if a given list `l` is `distinct` and a given function `f` is `injective`, then `map f l` produces a `distinct` list.

The `injective_map` property is provided as a Rocq`Parameter`. This means such a result is to be treated as an axiom: one can use it (and you should definitely use in one of your proofs!), but there is no need to prove it.

Exercise 6. If you want to prove `injective_map`, change it to a Lemma:

```
Lemma injective_map: ∀ [A B: Type] (f: A → B) (l: list A),
  distinct l → injective f → distinct (map f l).
```
Rocq

□

A OCaml Implementation of Proper Cuts

```OCaml
type 'a cut_list = ('a list * 'a list) list

let cons (x: 'a) (l: ('a list * 'a list)) : 'a list * 'a list =
  let (l1, l2) = l in
  (x :: l1, l2)

let rec proper_cuts (l: 'a list) : 'a cut_list =
  match l with
  | [] -> [([], [])]
  | x :: xs ->
    let pr = proper_cuts xs in
    let r = List.map (cons x) pr in
    ([], l) :: r
```