Coq でソートを実装

実装したソート

https://arxiv.org/abs/2110.01111

Wikipadiaには "I Can't Believe It Can Sort" として載っている。

```
def sort(A):
  for i in range(len(A)):
    for j in range(len(A)):
      if A[i] < A[j]:
        A[i], A[j] = A[j], A[i]

A = [1,5,6,2,7,3,4]
sort(A)
print(A) # [1,2,3,4,5,6,7]
```

仕組み

- 外側のループが `i` 周した時点で、左から `i` 個は整列済み。
- 1周目のループで最大値が `A[0]` に入る。
- 内側のループ(slide)は整列済みの部分列に `A[i]` を挿入する。
  - たとえば `[1, 5, 7] + [2]` を `[1, 2, 5] + [7]` にする。
  - `[7]` は最大値なので後半部分とは入れ替わらない。

`[1, 5, 6, 2, 7, 3, 4]` -> `[7, ^ 1, 5, 2, 6, 3, 4]`
-> `[1, 7, ^ 5, 2, 6, 3, 4]` -> `[1, 5, 7, ^ 2, 6, 3, 4]`
-> `[1, 2, 5, 7, ^ 6, 3, 4]` -> `[1, 2, 5, 6, 7, ^ 3, 4]`
-> `[1, 2, 3, 5, 6, 7, ^ 4]` -> `[1, 2, 3, 4, 5, 6, 7]`

関数型言語での実装が面倒（あるいは自分の実装が下手）

挿入ソートなどと違い、毎回リスト全体を触らないといけない

→ 対象となるリストを小さくしていくことが難しい。帰納的な実装に工夫がいる。

完成物

ソースコード

jsCoqなどで実行すると無事ソートであることが証明できる。

Coq のリストについて

```
Inductive list (A : Type) : Type :=
    | nil : list A
    | cons : A -> list A -> list A.
```

- 要素を1つ先頭に追加するときは `x :: xs`
- リストを連結するときは `l1 ++ l2`
- 空リストは `[]`, 1つの要素を持つリストは `[x]`

整列済み判定

```
Inductive sorted : list nat -> Prop :=
  | sorted_nil : sorted []
  | sorted_1 : forall x, sorted [x]
  | sorted_cons : forall {x y l},
      x <= y -> sorted (y :: l) -> sorted (x :: y :: l).
```

- 空リストは整列済み。
- 1つの要素を持つリストも整列済み。
- 先頭の要素以下の要素を付け加えたリストも整列済み。

置換判定

公式ライブラリのものを利用した。

```
Inductive Permutation : list A -> list A -> Prop :=
| perm_nil: Permutation [] []
| perm_skip x l l' : Permutation l l' -> Permutation (x::l) (x::l')
| perm_swap x y l : Permutation (y::x::l) (x::y::l)
| perm_trans l l' l'' :
    Permutation l l' -> Permutation l' l'' -> Permutation l l''.
```

- 空リストは空リストの置換。
- 先頭に同じ値を付け加えても置換。先頭2つ入れ替えても置換。
- 置換の置換は置換。

実装の工夫

- `i` の前半と後半で分割して2つのリストを持つようにした。
- `slide` 関数が内側のループを表す。
  先の例は `slide 2 [1;5;7]` を計算すると `(7, [1;2;5])` が帰ってくることに対応する（前後が逆で分かりにくい…）。
- `sort_kernel` 関数が実質全体のソートを表す。余計なものがくっついたりするが、 `sort_kernel (l1, l2)` によって `l1++l2` のソート結果が（分割されて）返ってくる。
- `naivesort l` が `sort_kernel (l, [])` に対応する。

実装の工夫その2

`sort_kernel`関数が停止することを追加で証明する必要があるが、その結果内部でCoqによる証明が勝手に追加され、扱いが面倒になる。

→ `Program`コマンドを利用して定義と同時に性質を証明するようにした（そのせいで受け渡す値に余計な証明がくっついている）。

抽出

このコードは（なぜか警告が出るが）Ocamlに抽出できる。
抽出したもの

感想

この上なくめんどくさいソートアルゴリズム。