Assignment 2:
Induction, Coinduction, and Recursion

15-312: Principles of Programming Languages (Fall 2023)

In this assignment, we'll explore *generic programming* with inductive and coinductive types, embedded in Standard ML. In functional programming lingo, this technique is often called "recursion schemes".[1] We will then consider how to formally prove code using inductive and coinductive types by induction and coinduction, respectively. Finally, we will briefly consider partiality in **PCF**.

[1] To learn more, see:

- Functional Pearl: Programming with Recursion Schemes (Wang and Murphy)
- Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire (Meijer, Fokkinga, and Paterson)
- An Introduction to Recursion Schemes (Thomson)

1 Type Operators

First, we will briefly introduce *type operators*, which wil allow us to share common utilities across implementations of inductive and coinductive types.

recursion-schemes/generic/positive-type-operator.sig

```
signature TYPE_OPERATOR =
sig
  type 't view (* parameter *)
end
```

Figure 1.1: Signature for type operators

In Fig. 1.1, we reproduce the typeclass `TYPE_OPERATOR`.[2] A type operator is simply a parameterized type `'t view`, which we will use to define the "shape" of the induction (or coinduction). On this assignment, we will use `signature POSITIVE_TYPE_OPERATOR`, which extends `TYPE_OPERATOR` slightly.[3]

recursion-schemes/experiments/type-ops.sml

```
structure NatOp =
struct
  datatype 't view = Zero | Succ of 't
sig
```


Figure 1.2: Type operator for natural numbers

Let's look at a simple example, the type operator for natural numbers, in Fig. 1.2. We define `'t view` to have two cases, `Zero` and `Succ`. However, notice that `'t view` is *not* recursive! In place of where we would like to "recur" (in the `Succ` case), we include the type variable `'t`. In Section 2, we will see how to use `NatOp` to define the natural numbers; then, `Zero` and `Succ` will behave somewhat like constructors.

[2] In **PFPL**, this is written $t\,.\,\tau$, where τ is the `view` type.

[3] The positivity requirement guarantees that `view` is "functorial", i.e., has a structure-preserving `map` function. For more details, see **PFPL**, Chapter 14.

2 Inductive Types

Now, we will consider inductive types. In our setup, we will define inductive types "generically" (in a common library) based around type operators. This will allow us to reuse infrastructure across all inductive types, including natural numbers, lists, and trees.

`recursion-schemes/generic/inductive.sig`

```
signature INDUCTIVE =
sig
  structure T: POSITIVE_TYPE_OPERATOR (* parameter *)
  type t (* abstract *)
  val FOLD: t T.view -> t
  val REC: ('rho T.view -> 'rho) -> t -> 'rho
end
```

Figure 2.1: Signature for inductive types

In Fig. 2.1, we show the signature for generalized inductive types. First, we have some type operator parameter `structure T`. We also have an abstract type `t` that will be the least fixed point of `T`; in other words, the "inductively iterated" version of `T`.[4] Finally, we have the expected operations: the generalized constructor `FOLD` and the recursor `REC`. We use `'rho`/ρ for the **r**esult type of the recursor.

IMPORTANT: To get points for these tasks, you must *not* use any explicit recursion in Standard ML (e.g., `fun`) or any built-in infinite datatypes (e.g., `int`). You should only use the provided inductive types and their recursors.

Throughout this section, you will work in `recursion-schemes/experiments/experiments.fun`. See `recursion-schemes/experiments/experiments.sig` for the signature to implement. You can find the relevant type operators in `recursion-schemes/experiments/type-ops.sml`.

2.1 Natural Numbers

Let us consider natural numbers, defined as a `structure Nat : INDUCTIVE where T = NatOp` (i.e., letting the `structure T` be `NatOp` from Fig. 1.2). There is a key difference between `NatOp` and `Nat`: the former describes a "template", whereas the second describes the inductive type of natural numbers. Using the two together, we may define:

$$\begin{aligned}
\texttt{nat} &\triangleq \texttt{Nat.t} \\
\texttt{z} &\triangleq \texttt{Nat.FOLD NatOp.Zero} \\
\texttt{s(}\,\texttt{n}\,\texttt{)} &\triangleq \texttt{Nat.FOLD (NatOp.Succ n)} \\
\texttt{iter}[\rho](\,\texttt{n}\,;\,\texttt{e0}\,;\,\texttt{x}\,.\,\texttt{e1}\,) &\triangleq \texttt{Nat.REC (fn NatOp.Zero => e0 | NatOp.Succ x => e1) n}
\end{aligned}$$

[4] In **PFPL**, Chapter 15, this is written $\mu(t\,.\,\tau)$.

Observe that the constructors `z` and `s( n )` are combined into one unified recursive constructor, `FOLD`, separating the inductive code on `Nat.t` from the sum type `'t NatOp.view`. Similarly, the base case `e0` and the inductive case `x . e1` are combined into the parameter of `REC`. Note that `x` has type ρ, not `nat`: the inductive result is put in all `'t` positions. This will generalize naturally to data structures with multiple recursive sub-components, such as trees, where each inductively-computed result will be in the corresponding `view` position.

Remark. One may be tempted to say, for example, `Nat.T.Zero` instead of `NatOp.Zero`; however, although it is known that `Nat.T = NatOp`, one quirk of Standard ML is that the constructors `Zero` and `Succ` are not available under `Nat.T` since `Nat.T : INDUCTIVE`.

```
type nat = Nat.t

val ZERO : nat        = Nat.FOLD N.Zero
val SUCC : nat -> nat = Nat.FOLD o N.Succ

val double : nat -> nat =
  Nat.REC
    (fn N.Zero   => ZERO
      | N.Succ r => SUCC (SUCC r))

val add : nat * nat -> nat =
  fn (m, n) =>
    Nat.REC
      (fn N.Zero   => n
        | N.Succ r => SUCC r)
      m
```


Figure 2.2: Simple functions on natural numbers, where `structure N = NatOp`

In Fig. 2.2, we define some basic functions on natural numbers using convenience functions `ZERO` and `SUCC`. The `double` code is straightforward; we assume that our inductive result has already been computed in the `Succ` case. In `add`, we elect (here, without loss of generality) to go by induction on `m`, using `n` in the base case.

Task 2.1 (15 pts). In `recursion-schemes/experiments/experiments.fun`, complete the definition of `NatUtil`. In particular:

- `exp2 n` should compute the `n` th power of 2.
- `halve n` should compute the floor of $\frac{\mathtt{n}}{2}$.
- `fib n` should compute the `n` th Fibonacci number F_n, where $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_n + F_{n+1}$.

Hint. You should be able to implement `exp2` in a straightforward manner using `double`. For `halve` and `fib`, you may wish to consider strengthening your inductive hypothesis, computing more than you need. You are welcome to use finite products and sums in Standard ML, such as booleans and pairs.

Testing If you open the SML/NJ REPL, you can test your code via `structure Experiments`, an instantiation of your `functor Experiments`. It is instantiated such that `Nat.t` is in fact `int`, allowing you to test easily.

```
smlnj -m sources.cm
- open Experiments;
- NatUtil.exp2 0;
val it = 1 : nat
- NatUtil.exp2 10;
val it = 1024 : nat
- NatUtil.halve 312;
val it = 156 : nat
- NatUtil.halve 313;
val it = 156 : nat
- NatUtil.fib 14;
val it = 377 : nat
```


2.2 Lists

Now, let's consider another inductive types: lists of natural numbers.


```
structure ListOp =
  struct
    type element = Nat.t
    datatype 't view = Nil | Cons of element * t
  end
```


Figure 2.3: Type operator for lists

Just as with natural numbers, we first define the "type operator", concisely reproduced in Fig. 2.3. Then, we include a `structure List : INDUCTIVE where T = ListOp`.

We show the implementation of some sample list functions in Fig. 2.4.

Task 2.2 (10 pts). In `recursion-schemes/experiments/experiments.fun`, implement:

- `ListUtil.sum l` to compute the sum of `l` (using `NatUtil.add`).
- `ListUtil.filter p l` to compute a list of the elements in `l` satisfying and not satisfying `p`, in the order they appear in `l`. It should behave like `List.filter` from the Standard ML Basis Library.

Task 2.3 (10 pts). In `recursion-schemes/experiments/experiments.fun`, implement the function `ListUtil.reverse` such that `reverse l` reverses the list `l`, a la `List.rev`. Your imple-

```
type list = List.t

val length : list -> nat =
  List.REC
    (fn L.Nil => ZERO
      | L.Cons (_, n) => SUCC n)

val map : (element -> element) -> list -> list =
  fn f =>
    List.REC
      (fn L.Nil            => List.FOLD L.Nil
        | L.Cons (x, xs) => List.FOLD (L.Cons (f x, xs)))
```

Figure 2.4: Sample functions on lists of natural numbers, where `structure L = ListOp`

mentation should run in linear time in terms of the length of the list; a super-linear runtime solution will earn you partial credit.

Hint. You may find it useful to recall the recursive implementation of list reverse:


```
fun reverseHelper nil       acc = acc
  | reverseHelper (x :: xs) acc = reverseHelper xs (x :: acc)

fun reverse l = reverseHelper l nil
```


You may want to refresh your memory on CPS from 15-150; consider accumulating a function.

Testing Once again, you can test your code via `structure Experiments`. It instantiates your `functor Experiments` such that `List.t` is in fact `int list`, allowing you to test easily.

```
smlnj -m sources.cm
- open Experiments;
- ListUtil.sum [1, 5, 3, 1, 2];
val it = 12 : int
- ListUtil.filter (fn x => x <= 2) [1, 5, 3, 1, 2];
val it = [1,1,2] : int list
```

2.2.1 Derived Forms

Thus far, the functions we have asked you to consider have been fairly amenable to simple inductive implementation. However, not all code is so straightforward! For example, we often wish to pattern match on the outer layer of a list. Additionally, we sometimes want immediate access to our predecessor, as included natively in the **T** recursor. While neither of these operations are present as primitives, we can implement them as *derived forms*.

```
fun stripLeading p nil = nil
  | stripLeading p (x :: xs) =
      if p x
        then stripLeading p xs
        else x :: xs

val stripLeading : (element -> bool) -> list -> list =
  fn p =>
    ListUtil.REC'
      (fn ListOp.Nil => List.FOLD ListOp.Nil
        | ListOp.Cons (x, (res, xs)) =>
            if p x
              then res
              else List.FOLD (ListOp.Cons (x, xs)))
```

Figure 2.5: Sample usage of `ListUtil.REC'`

Task 2.4 (10 pts). In `recursion-schemes/experiments/experiments.fun`, implement the function `ListUtil.UNFOLD : list -> list ListOp.view`, which will allow us to unwrap one layer of a list and pattern match immediately on `ListOp.Nil` and `ListOp.Cons`. Concretely, `UNFOLD (List.FOLD v)` should evaluate to `v : list ListOp.view`.

Hint. Consider going by induction (i.e., `REC`).

Task 2.5 (10 pts). In `recursion-schemes/experiments/experiments.fun`, implement the function `ListUtil.REC' : (('rho * list) ListOp.view -> 'rho) -> list -> 'rho`, which will be similar to `REC` while also providing us with the list tail at each layer.

The usage of `REC'` shown in Fig. 2.5 should be identical to the corresponding recursive Standard ML code.

Hint. Consider "strengthening your IH": in addition to computing not only the desired result, you may wish to inductively reconstruct the list itself.

2.2.2 Insertion Sort

Let's implement insertion sort inductively. First, recall its usual implementation in Fig. 2.6.

Task 2.6 (20 pts). In `recursion-schemes/experiments/experiments.fun`, implement the structure `InsertionSort`. In particular, you should define `insert` and `sort` to mirror the above definitions. The comparison function `<=` is provided at the top of the file.

Hint. In the second case of `insert`, you sometimes return `x :: y :: ys`, where `ys` is unchanged. Which of the previously-defined derived forms might help you accomplish this?

```
fun insert (x, nil      ) = x :: nil
  | insert (x, y :: ys) =
      if x <= y
        then x :: y :: ys
        else y :: insert (x, ys)

fun sort nil         = nil
  | sort (x :: xs) = insert (x, sort xs)
```

Figure 2.6: Insertion sort, recursively

2.2.3 Merge Sort

While insertion sort is structurally recursive, one might wonder: how could we implement an algorithm like merge sort, which recurs on lists which are not the immediate tail? Recall its implementation in Fig. 2.7.

Consider `sort`: in the last case, we recursively sort `l1` and `l2`. To implement this behavior using the recursor, we use a clever trick: we go by induction not on the list itself, but on a "clock" indicating the desired recursion depth.

This approach is shown in Fig. 2.8. Here, we recursively compute a function of type `list -> list`, with a specification that at inductive layer k, we produce a correct sorting function for lists of length up to 2^k.

1. In the base case $k = 0$, the identity function suffices, since it sorts lists of length 0 and 1.
2. In the inductive case $k = k' + 1$, we implement the usual algorithm, using an inductively-computed `f` capable of sorting lists of size $2^{k'}$ on the smaller lists.

We start the recursion with $k =$ `length l` and apply the resulting function to `l`.

Task 2.7 (5 pts). In the above code, we start with $k =$ `length l`; while this is a sufficiently large value of k to correctly sort, it is far larger than necessary, causing the code to recur on empty and singleton lists repeatedly. What would be an efficient starting value for k in terms of `l`? You may ignore small constants/off-by-one issues. Explain briefly (in 1-2 sentences).

Task 2.8 (20 pts). In `recursion-schemes/experiments/experiments.fun`, implement `merge` in the structure `MergeSort`.

Hint. You will almost certainly need to use the aforementioned technique, going by induction on a clock rather than by structural induction on one of the lists. Additionally, you may find one of the derived forms implemented earlier particularly useful.

```
fun split nil        = (nil, nil)
  | split (x :: xs) =
      let
        val (l1, l2) = split xs
      in
        (l2, x :: l1)
      end

fun merge (nil     , l2      ) = l2
  | merge (l1      , nil     ) = l1
  | merge (x :: xs, y :: ys) =
      if x <= y
        then x :: merge (xs, y :: ys)
        else y :: merge (x :: xs, ys)

fun sort nil = nil
  | sort (x :: nil) = x :: nil
  | sort (x :: xs) =
      let
        val (l1, l2) = split (x :: xs)
      in
        merge (sort l1, sort l2)
      end
```


Figure 2.7: Merge sort, recursively


```
val sort : list -> list =
  fn l =>
    Nat.REC
      (fn N.Zero => (fn l => l)
        | N.Succ f =>
            fn l =>
              let
                val (l1, l2) = split l
              in
                merge (f l1, f l2)
              end)
      (ListUtil.length l)
      l
```

Figure 2.8: Merge sort, inducting on a “clock”

3 Coinductive Types

Let us now shift our attention to coinductive types. We will reuse the "type operator" infrastructure,

`recursion-schemes/generic/coinductive.sig`

```
signature COINDUCTIVE =
sig
  structure T: POSITIVE_TYPE_OPERATOR (* parameter *)
  type t (* abstract *)
  val GEN: ('sigma -> 'sigma T.view) -> 'sigma -> t
  val UNFOLD: t -> t T.view
end
```

Figure 3.1: Signature for coinductive types

considering a new `COINDUCTIVE` signature in Fig. 3.1. As in `INDUCTIVE`, we have a type operator parameter `structure T`. We also have an abstract type `t`, but this time, it will be the *greatest* fixed point of `T`, allowing for infinite elements.[5] Finally, we have the expected operations: the generator `GEN` and the generalized destructor `UNFOLD`. We use `'sigma` / σ for the state type of the generator.

Remark. Inductive and coinductive types are "dual" in a mathematically precise way. You may have already noticed the similarity between `INDUCTIVE` and `COINDUCTIVE`: they are identical, except with (some) function arrows reversed!

IMPORTANT: To get points for these tasks, you must *not* use any explicit recursion in Standard ML (e.g., `fun`) or any built-in infinite datatypes (e.g., `int`). You should only use the provided (co)inductive types and their recursors/generators.

3.1 Streams

Using the `ListOp` type operator from Section 2.2, we can define potentially-infinite streams via a `structure Stream : COINDUCTIVE where T = ListOp`.

When working with inductive types, we used `FOLD` to create finite data structures layer by layer, allowing `REC` to "tear down" the entirety of the data structure. Dually, when working with coinductive types, we will create (potentially) infinite data structures in their entirety using `GEN`, allowing `UNFOLD` to make finitely many observations about the data structure layer by layer.

In Fig. 2.4, we implemented `map`, which applies a function to every element of a list, by recursing over an existing list. We implement an analogous `map` function on streams in Fig. 3.2. In particular, we use `Stream.GEN` with an internal state of type `stream`; at each layer, we mirror the structure of the state stream, applying the function `f` to each element.

[5] In **PFPL**, Chapter 15, this is written $\nu(t\,.\,\tau)$.

```
type stream = Stream.t

val map : (element -> element) -> stream -> stream =
  fn f =>
    Stream.GEN
      (fn s =>
        case Stream.UNFOLD s of
          L.Nil           => L.Nil
        | L.Cons (x, xs) => L.Cons (f x, xs))
```

Figure 3.2: Simple function on streams of natural numbers, where `structure L = ListOp`

Task 3.1 (10 pts). In `recursion-schemes/experiments/experiments.fun`, implement the remaining functions in `structure StreamUtil`.

- `fromList l` should produce a (finite) stream which produces the elements from `l` in order.
- `zipWith f (s1, s2)` should provide a stream which produces a stream consisting of the elements of `s1` and `s2` combined pairwise with `f`, a la `ListPair.map`. If either stream terminates, the entire stream should terminate.

Hint. You may find `ListUtil.UNFOLD` helpful in `fromList`.

Testing You can, once again, test your code via `structure Experiments`. We provide a special implementation of streams which makes the front visible.


```
smlnj -m sources.cm
- open Experiments;
- StreamUtil.fromList [1, 5, 3, 1, 2];
val it = HIDE ([1,5,3,1,2],NONE) : stream
- val squares = Stream.GEN (fn i => ListOp.Cons (i * i, i + 1)) 0;
val squares = HIDE ([0,1,4,9,16,25,36,...],SOME fn) : Stream.t
- StreamUtil.map (fn i => i + 1) squares;
val it = HIDE ([1,2,5,10,17,26,37,...],SOME fn) : stream
- StreamUtil.zipWith (op +) (squares, StreamUtil.fromList [1, 5,
   3, 1, 2]);
val it = HIDE ([1,6,7,10,18],NONE) : stream
```

In Section 2.2, we considered `map` and `filter` on lists. However, while we wrote `map` on streams, it is impossible to write an analogous `filter` function on streams using only the `COINDUCTIVE` interface.

Task 3.2 (5 pts). Briefly explain why it is impossible to write a `filter` function on (coinductively-defined) streams.

Hint. Try writing it, and see where you get stuck!

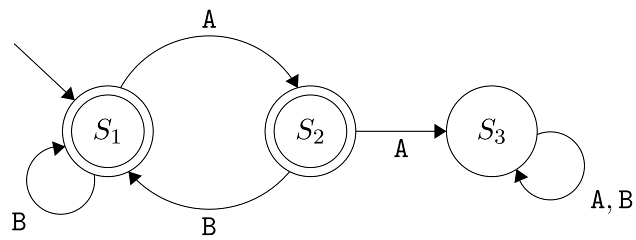

Figure 3.3: State diagram for machine $M_{\neg\texttt{AA}}$. For simplicity, we use `B` to represent non-`A` characters.

Remark. It *is* possible to write a `filter` function on streams using general recursion. (In fact, you may recall this from 15-150.)

3.2 Automata

Using our framework for coinductive types, we can elegantly describe and understand strings and automata.

An *automaton* "reads" through a string character by character and decides whether it should *reject* or *accept* the string. Every automaton works on some set of internal states, each of which is either *accepting* or *rejecting*. An automaton starts in one state; when it reads a character from the string, it makes a transition to a new state according to the character it just read. After it processed all characters in the string, it accepts the string if and only if it ends up in an accepting state.

As an example, the automaton $M_{\neg\texttt{AA}}$ in Fig. 3.3 accepts a string iff it does not contain consecutive `A`s. Here S_1 is the starting state and S_1 and S_2 are accepting states. It accepts strings `""`, `"B"`, `"ABBA"`, `"ABA"`, but not strings `"AA"`, `"BAA"`, `"BBABAAB"`. $M_{\neg\texttt{AA}}$ has three states. It starts in S_1. Its accepting states include S_1 and S_2.

Now, we shall choose representations of strings and automata.

1. Strings will be an inductive type, defineed as lists of characters. Specifically, `StringOp` will be equivalent to `ListOp` from Fig. 2.3 but with `type element = char`.

2. Automata will be a coinductive type based around type operator `AutomatonOp` in Fig. 3.4. In other words, an automaton will be a `bool`, saying whether or not the current state is "accepting" or not, and a transition function from a `char` to another automaton. Notice that an automaton behaves much like a stream, but with one tail available for each `char`.

```
structure AutomatonOp =
  struct
    type 't view = bool * (char -> 't)
  end
```

Figure 3.4: Type operator for automata

First, we will define a function `run : automaton -> string -> bool`, allowing us to run an automaton on a string.

Task 3.3 (15 pts). In `recursion-schemes/experiments/experiments.fun`, implement the `run` function in `structure AutomatonUtil` according to the informal specification above. Your automaton should read from the "end" of the string first, which will visually be at the beginning.

Testing You can test your code via `structure Experiments`, where we implement `String` via built-in strings for convenience. In the starter code, we define $M_{\neg \texttt{AA}}$ as `notConsecutiveA`. Additionally, we provide an example `abc`, the automaton that accepts only the string `"ABC"`.

```
smlnj -m sources.cm
- open Experiments;
- open AutomatonUtil;
- run notConsecutiveA "ABBA";
val it = true : bool
- run notConsecutiveA "BBABAAB";
val it = false : bool
- run abc "ABC";
val it = true : bool
- run abc "ABCD";
val it = false : bool
```


Now, to implement some more automata!

Task 3.4 (15 pts). In `recursion-schemes/experiments/experiments.fun`, implement:

- `endsWithA : automaton` such that `run endsWithA` accepts all strings that end with `"A"`.
- `abStar : automaton` such that `run abStar` accepts the strings in language:

$$\texttt{"AB"}^{\star} = \{\texttt{""}, \texttt{"AB"}, \texttt{"ABAB"}, \texttt{"ABABAB"}, \ldots\}$$

- `either : automaton * automaton -> automaton` such that `run (either (a1, a2))` accepts the strings that at least one of `a1` and `a2` accept.

You are welcome to define a custom (finite) datatype if you wish, as in examples `notConsecutiveA` and `abc`.

```
smlnj -m sources.cm
- open Experiments;
- open AutomatonUtil;
- run endsWithA "BA";
val it = true : bool
- run endsWithA "AB";
val it = false : bool
- run endsWithA "";
```

```
val it = false : bool
- run abStar "";
val it = true : bool
- run abStar "AB";
val it = true : bool
- run abStar "ABA";
val it = false : bool
- run (either (notConsecutiveA, endsWithA)) "AB";
val it = true : bool
- run (either (notConsecutiveA, endsWithA)) "BBAA";
val it = true : bool
- run (either (notConsecutiveA, endsWithA)) "BBAAB";
val it = false : bool
```

4 Proof by Induction and Coinduction

Inductive and coinductive types are, unsurprisingly, conducive to proof by induction and coinduction, respectively. In this part of the assignment, we will prove properties about operations on inductive and coinductive data.

4.1 Induction

We define list length and append functions as follows, using the infrastructure described in Section 2.2:

```
structure N = NatOp
structure L = ListOp

type nat = Nat.t
type list = List.t

val ZERO = Nat.FOLD N.Zero
val SUCC = Nat.FOLD o N.Succ

val length : list -> nat =
  List.REC
    (fn L.Nil => ZERO
      | L.Cons (_, n) => SUCC n)

val append : list * list -> list =
  fn (l1, l2) =>
    List.REC
      (fn L.Nil => l2
        | L.Cons (x, l) => List.FOLD (L.Cons (x, l)))
    l1

val add : nat * nat -> nat =
  fn (m, n) =>
    Nat.REC
      (fn N.Zero   => n
        | N.Succ r => SUCC r)
      m
```


The definition of lists as an inductive type tells us that:

`List.REC f (List.FOLD L.Nil)` $=$ `f L.Nil`

`List.REC f (List.FOLD (L.Cons (x, xs)))` $=$ `f (L.Cons (x, List.REC f xs))`

Since `list` is an inductive type, we can rigorously define what it means to go by induction on a value of type `list`. The induction principle works just like `REC`, “tearing down” a list to get a desired result in each case.

Definition 4.1 (Induction Principle for `list`)**.** Consider some property on lists, $P(-)$.
If it is the case that:

> If we assume that for all values `v : list L.view`, either:
>
> 1. `v` $\triangleq$ `L.Nil`
> 2. `v` $\triangleq$ `L.Cons (x, xs)`, where $P($ `xs` $)$
>
> then, $P($ `List.FOLD v` $)$ holds.

then for all `l : list`, $P($ `l` $)$.

For the rest of the problem, we will abbreviate `List.FOLD` as `FOLD` and `List.REC` as `REC`.

Let's use the induction principle to prove a simple theorem.

Theorem 4.2. *For all values* `l : list`*,* `append (l, FOLD L.Nil)` $=$ `l`*.*

Proof. We use the induction principle for `list`, where

$$P(\texttt{l}) \triangleq \texttt{append (l, FOLD L.Nil)} = \texttt{l}$$

Let `v : list L.view` be arbitrary, and assume that either

1. `v` $\triangleq$ `L.Nil`
2. `v` $\triangleq$ `L.Cons (x, xs)`, where `append (xs, FOLD L.Nil)` $=$ `xs`

It remains to show that `append (FOLD v, FOLD L.Nil)` $=$ `FOLD v`.

We go by cases on the assumption:

1. Suppose `v` $\triangleq$ `L.Nil`. Then:


```
  append (FOLD v, FOLD L.Nil)
= append (FOLD L.Nil, FOLD L.Nil)
= REC (fn L.Nil => l2 | L.Cons (x, l) => ...) (FOLD L.Nil)
= FOLD L.Nil                                                   (REC law)
```

   as desired.

2. Suppose `v` $\triangleq$ `L.Cons (x, xs)`, where `append (xs, FOLD L.Nil)` $=$ `xs`. Then:

```
  append (FOLD v, FOLD L.Nil)
= append (FOLD (L.Cons (x, xs)), FOLD L.Nil)
= REC (fn L.Nil => l2 | L.Cons (x, l) => ...) (FOLD (L.Cons (x, xs)))
= FOLD (L.Cons (x, append (xs, FOLD L.Nil)))                    (REC law)
= FOLD (L.Cons (x, xs))                                         (IH assumption)
```

   as desired.

□

Task 4.1 (15 pts). Prove by induction that for all values `l1, l2 : list`,

$$\texttt{length (append (l1, l2))} = \texttt{add (length l1, length l2)}$$

Hint. Which list should you go by induction on? Your proof should exactly mirror the code.

4.2 Coinduction

Consider the following type operator for infinite streams of natural numbers:

```
structure InfStreamOp =
  struct
    type 't view = nat * 't
  end
```

Let `structure InfStream : COINDUCTIVE where T = InfStreamOp`. We will freely assume arithmetic facts about `nat`.

We define `nats` and `evens`, the (infinite) streams of natural numbers and even numbers. Then, we define `inc` and `double`, which increment and double streams element-wise.


```
type infstream = InfStream.t

val nats : nat -> infstream =
  InfStream.GEN (fn n => (n, 1 + n))

val evens : nat -> infstream =
  InfStream.GEN (fn n => (2 * n, 1 + n))

val inc : infstream -> infstream =
  Stream.GEN
    (fn s =>
      let val (hd, tl) = InfStream.UNFOLD s
      in (1 + hd, tl) end)

val double : infstream -> infstream =
  Stream.GEN
    (fn s =>
      let val (hd, tl) = InfStream.UNFOLD s
      in (2 * hd, tl) end)
```

Clearly, there should be some relationship between these definitions. For example, we would expect that `inc (nats n)` is "equal to" `nats (1 + n)`. However, what does equality of infinite streams mean? We define a binary relation `s1` $\equiv$ `s2` as follows, intended to mean that `s1` and `s2` are equal, via the following coinduction principle. The coinduction principle works just like `GEN`, "building up" a stream by showing that a "state" is preserved.

Definition 4.3 (Coinduction Principle for Equality of Infinite Streams)**.** Consider some relation on streams, $R(-,-)$. If it is the case that:

If we assume $R($`s1`, `s2`$)$, then we have `n1` $=$ `n2` and $R($`s1'`, `s2'`$)$, where:

`InfStream.UNFOLD s1` $\triangleq$ `(n1, s1')`

`InfStream.UNFOLD s2` $\triangleq$ `(n2, s2')`

then if $R($`s1`, `s2`$)$, we have `s1` $\equiv$ `s2`.

Remark. $R(-,-)$ is like a "loop invariant", showing that `s1` and `s2` stay related at each iteration. Such a relation is also called a *bisimulation relation*. To proving two streams are equivalent, one must find a relation that is preserved by stream unfolding.

Remark. Equality of infinite streams is similar to function extensionality: it says that two streams are equal as long as they *behave* the same way, regardless of their implementation.

The definition of streams as a coinductive type tells us that:

`InfStream.UNFOLD (InfStream.GEN f x)` $\equiv$ `(n, InfStream.GEN f x')`

where `(n, x')` $=$ `f x`.

For the rest of the problem, we will abbreviate `InfStream.GEN` as `GEN` and `InfStream.UNFOLD` as `UNFOLD`. Let's use the coinduction principle to prove a simple theorem.

Theorem 4.4. *`inc (nats 0)`* $\equiv$ *`nats 1`*.

Proof. We use the coinduction principle for $\equiv$.

We choose $R(-,-)$ to relate all pairs (`inc (nats n)`, `nats (1 + n)`); in set-theoretic notation, $R = \{($`inc (nats n)`, `nats (1 + n)`$) \mid$ `n : nat`$\}$.[6]

First, assume that $R($`inc (nats n)`, `nats (1 + n)`$)$ for an arbitrary `n`. Then:

```
UNFOLD (inc (nats n))
= let val (hd, tl) = UNFOLD (nats n) in (1 + hd, inc tl) end          (GEN law)
= let val (hd, tl) = (n, nats (1 + n)) in (1 + hd, inc tl) end        (GEN law)
= (1 + n, inc (nats (1 + n)))
```

```
UNFOLD (nats (1 + n))                                                 (GEN law)
= (1 + n, nats (2 + n))
```

[6] Observe that R is stronger than the desired theorem! Choosing this stronger R is like finding a stronger loop invariant. The process is called "strengthening the coinductive hypothesis", dual to the more familiar strengthening of the inductive hypothesis.

Observe that `1 + n` $=$ `1 + n`, as desired. Additionally, $R($ `inc (nats (1 + n))` , `nats (2 + n)` $)$. Therefore, the assumption holds!

Now, it remains to show that $R($ `inc (nats 0)` , `nats 1` $)$: of course, this is trivially true, considering `n` $=$ `0`. Thus, `inc (nats 0)` $\equiv$ `nats 1`. $\square$

Task 4.2 (15 pts). Prove by coinduction that `double (nats 0)` $\equiv$ `evens 0`.

5 General Recursion in **PCF**

In **PCF**, we have general recursion. This should be rather familiar: the vast majority of widely-used programming languages, including Standard ML, come equipped with general recursion.

So far, we have expressed the dynamics of programming languages via transition rules; in **PCF**, we maintain this approach. Even though **PCF** is partial (not every expression terminates), our techniques are unaffected!

We implemented the statics already in `lang-pcf/statics-pcf.sml`.

The expected Progress theorem holds for **PCF**:

Theorem 5.1 (Progress)**.** *If $e : \tau$, then either:*

- *there exists some e' such that $e \longmapsto e'$, or*
- *e val.*

As usual, its proof requires the use of a canonical forms lemma:

Lemma 5.2 (Canonical Forms)**.** *If e val and $e : \tau_1 \rightharpoonup \tau_2$, then $e = \texttt{fun}[\,\tau_1\,;\tau_2\,](\,f\,.\,x\,.\,e_2\,)$ for some $\tau_1, \tau_2, f, x, e_2$.*

Task 5.1 (10 pts)**.** Prove Theorem 5.1 for function values and function application.

Preservation holds for **PCF**, as well.

Task 5.2 (10 pts)**.** In `lang-pcf/dynamics-pcf.sml`, implement the dynamics of **PCF** specified in Appendix B. The state implementation, `StatePCF`, is identical to the state from the previous assignment, `State.Step e` and `State.Val v`, other than the fact that it uses **PCF** expressions. You can find the ABT signature for **PCF** in the expected file, `lang-pcf/pcf.abt.sig`.

In a total language (including languages with inductive and coinductive types), we are guaranteed that all programs terminate. In a partial language like **PCF**, on the other hand, some expressions do not terminate.

Task 5.3 (5 pts)**.** Give one plausible reason a user may prefer a *total* language and one plausible reason a user may prefer a *partial* language. Justify your reasons briefly (1-2 sentences each).

Testing You can test in the `InterpreterPCF` REPL:

```
smlnj -m lang-pcf/sources.cm

- InterpreterPCF.repl ();
-> (fun f (x : nat) : nat is s x) 5;
```

```
(Ap ((Fun ((Nat, Nat), (f21 . (x22 . (S x22))))), (S (S (S (S (S
   Z)))))))
Type: Nat
Evaluating... val (S (S (S (S (S (S Z))))))
```

Some simple examples of the concrete syntax are available in `lang-pcf/tests/tests.pcf`.

A Statics of PCF

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Var}$$

A.1 Natural Numbers

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}\ \textsc{Z} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(\, e \,) : \mathtt{nat}}\ \textsc{S} \qquad \frac{\Gamma \vdash e : \mathtt{nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}[\, e_0 \,;\, x \,.\, e_1 \,](\, e \,) : \tau}\ \textsc{Ifz}$$

A.2 Partial Functions

$$\frac{\Gamma, f : \tau_1 \rightharpoonup \tau, x : \tau_1 \vdash e : \tau}{\Gamma \vdash \mathtt{fun}[\, \tau_1 \,;\, \tau \,](\, f \,.\, x \,.\, e \,) : \tau_1 \rightharpoonup \tau}\ \textsc{Fun} \qquad \frac{\Gamma \vdash e : \tau_1 \rightharpoonup \tau \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{ap}(\, e \,;\, e_1 \,) : \tau}\ \textsc{Ap}$$

B Dynamics of **PCF**

B.1 Natural Numbers

$$\frac{}{\texttt{z}\ \mathsf{val}}\ \textsc{Z-Val} \qquad \frac{e \longmapsto e'}{\texttt{s}(\,e\,) \longmapsto \texttt{s}(\,e'\,)}\ \textsc{S-Step} \qquad \frac{e\ \mathsf{val}}{\texttt{s}(\,e\,)\ \mathsf{val}}\ \textsc{S-Val}$$

$$\frac{e \longmapsto e'}{\texttt{ifz}[\,e_0\,;x\,.\,e_1\,](\,e\,) \longmapsto \texttt{ifz}[\,e_0\,;x\,.\,e_1\,](\,e'\,)}\ \textsc{Ifz-Step} \qquad \frac{\texttt{z}\ \mathsf{val}}{\texttt{ifz}[\,e_0\,;x\,.\,e_1\,](\,\texttt{z}\,) \longmapsto e_0}\ \textsc{Ifz-Z}$$

$$\frac{\texttt{s}(\,e\,)\ \mathsf{val}}{\texttt{ifz}[\,e_0\,;x\,.\,e_1\,](\,\texttt{s}(\,e\,)\,) \longmapsto \{e/x\}e_1}\ \textsc{Ifz-S}$$

B.2 Partial Functions

$$\frac{}{\texttt{fun}[\,\tau_1\,;\tau\,](\,f\,.\,x\,.\,e\,)\ \mathsf{val}}\ \textsc{Fun-Val} \qquad \frac{e \longmapsto e'}{\texttt{ap}(\,e\,;e_1\,) \longmapsto \texttt{ap}(\,e'\,;e_1\,)}\ \textsc{Ap-Step}$$

$$\frac{e\ \mathsf{val} \qquad e_1 \longmapsto e_1'}{\texttt{ap}(\,e\,;e_1\,) \longmapsto \texttt{ap}(\,e\,;e_1'\,)}\ \textsc{Ap-Step1}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{ap}(\,\texttt{fun}[\,\tau_1\,;\tau\,](\,f\,.\,x\,.\,e\,)\,;e_1\,) \longmapsto \{\texttt{fun}[\,\tau_1\,;\tau\,](\,f\,.\,x\,.\,e\,), e_1/f, x\}e}\ \textsc{Ap}$$