CIS 7000-1 Homework 6

NAME: FILL IN HERE

November 19, 2025

1 Monads are Monoids in the Category of Endofunctors

(The starter code for this problem is in the file: rocq/control/monads.v.)

Recall the definition of *difference lists* or `dlists` from class. The key idea of this data structure is that it abstracts over "what should be in the rest of the list". This allows an efficient implementation of list append as function composition.

```
Definition dlist (A : Type) := list A → list A.
Definition dnil {A} : dlist A := fun k ⇒ k.
Definition dcons {A} : A → dlist A → dlist A := fun x dl ⇒ fun k ⇒ x :: dl k.
Definition dapp {A} : dlist A → dlist A → dlist A := fun f g ⇒ fun x ⇒ (f (g x)).
Definition list_of_dlist {A} : dlist A → list A := fun dl ⇒ dl [].
```

1. Show that difference lists are monoids. In other words, prove the following three properties.

   ```
   Lemma dlist_left_id {A} {dl : dlist A} :
     dapp dnil dl = dl.
   Lemma dlist_right_id {A} {dl : dlist A} :
     dapp dl dnil = dl.
   Lemma dlist_assoc {A}{dl1 dl2 dl3 : dlist A} :
     dapp (dapp dl1 dl2) dl3 = dapp dl1 (dapp dl2 dl3).
   ```

2. Consider this reimplementation of the list reverse function. If you unfold the definitions, you will see that it is the same as the implementation of list reversal using an accumulator.

   ```
   Fixpoint reverse_dlist {A} (xs : list A) : dlist A :=
     match xs with
     | nil ⇒ dnil
     | y :: ys ⇒ dapp (reverse_dlist ys) (dcons y dnil)
     end.
   ```

   Prove that this implementation of reverse is the same as the usual definition (which is as above but uses lists instead of dlists).

   ```
   Lemma reverse_dlist_spec {A} (xs : list A) : List.rev xs = list_of_dlist (reverse_dlist xs).
   ```

3. The *continuation monad* is a way of representing computations that abstract over "what to do next". This allows a convenient way of defining code in continuation passing style.

   Here is how we can define the continuation monad in Rocq.

   ```
   Definition M (K A : Type) := (A → K) → K.
   Definition ret {A K} : A → M K A := fun x k ⇒ k x.
   Definition bind {K A B} : M K A → (A → M K B) → M K B :=
     fun m1 m2 ⇒ fun (k : B → K) ⇒ m1 (fun a ⇒ m2 a k).
   ```

We'll also introduce standard notation for the bind operation.

```
Infix ">=" := bind (at level 70).
Notation "x ← m1 ;; m2" := (bind m1 (fun x ⇒ m2)) (at level 70).
```

Prove that M K is a monad by showing the three monad laws.

```
Lemma M_left_id {K A B}{x:A}{h : A → M K B} : ret x >=h = h x.
Lemma M_right_id {K A}{x:A}{m : M K A} : m >=ret = m.
Lemma M_assoc {K A B C}{m : M K A}{g : A → M K B}{h : B → M K C} :
  ((m >=g) >=h) = (m >=(fun (x : A) ⇒ g x >=h)).
```

4. We can rewrite the reverse function using the continuation monad as follows:

```
Fixpoint reverse_cps {K A} (xs : list A) : M K (list A) :=
  match xs with
  | nil ⇒ ret nil
  | y :: ys ⇒ zs ← reverse_cps ys ;; ret (zs ++[y])
  end.
```

Prove that this version of reverse is equivalent to the usual definition.

```
Lemma reverse_cps_spec {A} (xs: list A) : rev xs = reverse_cps xs (fun x ⇒ x).
```

2 Continuation-Passing Style Translation

In this problem you will work with the CPS translation that we discussed in class. The core language is the stack based language with **letcc**, **throw**, **cont**, and **exit**.

For reference, the files `homework/rocq/hw5/control/letcc.v` and `homework/rocq/hw5/control/cps.v` contains definitions to help you get started.

Recall the theorem that states that the CPS translation is type preserving.

Lemma 2.1 (CPS translation is type preserving)**.** Suppose $\Gamma \sim \Delta$.

1. If $\Gamma \vdash v \in \tau$ then $\Delta \vdash \mathcal{V}(v) \in \mathcal{T}(\tau)$

2. If $\Gamma \vdash e \in \tau$ and $\Delta \vdash w \in \mathcal{C}(\tau)$ then $\Delta \vdash \mathcal{E}(e)_w \in \mathbf{Void}$

3. If $\vdash s \in \tau_1 \rightsquigarrow \tau$ then $\vdash \mathcal{S}(s) \in \mathcal{C}(\tau)$

This lemma depends on the definitions of the type, term, value and stack translations, as well as the relation $\Gamma \sim \Delta$, which are available in the lecture notes.

1. Prove this lemma. Note you don't need to type set all of the cases here: just pick a small number that you believe are representative.

2. (Optional) This translation is not the only possible CPS translation. Pick a different one, define how it works and prove the analogous lemma that states that the translation preserves types. You don't need to include the full language that we include in the lecture notes. At a minimum, your translation should work for the core lambda calculus: variables, functions, applications, ret and let. (Do *not* try to prove a simulation relation for the translation that you have picked.)

   For example, the translation that Pottier uses in his paper produces more efficient output by generalizing the continuation argument to the term translation. (His version is based on a translation by Danvy and Filinski from 1992). Or, you could look up Plotkin's original call-by-value translation (in his 1975 paper), or one created by Sabry and Wadler in 1997. For a more logical point of view, Sørensen and Urzyczyn in "Lecture Notes on the Curry Howard Isomorphism", present a different translation based on the double negation translation studied since the 1930s.