CSE301 Functional Programming, Fall 2023

In-class exam

(solutions)

This exam consists of five parts that can be solved in any order.

1 First-order data types

For the first two questions, consider the following data type of bit strings:

$$\textbf{data}\ \mathit{Bits} = \mathit{BNil} \mid \mathit{B0}\ \mathit{Bits} \mid \mathit{B1}\ \mathit{Bits}$$

We interpret bit strings as representing natural numbers in binary like so:

$$\begin{array}{l} \mathit{toInt} :: \mathit{Bits} \to \mathit{Int} \\ \mathit{toInt}\ \mathit{BNil} = 0 \\ \mathit{toInt}\ (\mathit{B0}\ x) = 2 * \mathit{toInt}\ x \\ \mathit{toInt}\ (\mathit{B1}\ x) = 2 * \mathit{toInt}\ x + 1 \end{array}$$

Question 1.1. Write a function $\mathit{inc} :: \mathit{Bits} \to \mathit{Bits}$ that increments the bit string representing a number. It should satisfy the property that $\mathit{toInt}\ (\mathit{inc}\ \mathit{bs}) = 1 + \mathit{toInt}\ \mathit{bs}$.

Solution :

$$\begin{array}{l} \mathit{inc} :: \mathit{Bits} \to \mathit{Bits} \\ \mathit{inc}\ \mathit{BNil} = \mathit{B1}\ \mathit{BNil} \\ \mathit{inc}\ (\mathit{B0}\ x) = \mathit{B1}\ x \\ \mathit{inc}\ (\mathit{B1}\ x) = \mathit{B0}\ (\mathit{inc}\ x) \end{array}$$

We can verify by induction that $\mathit{toInt}\ (\mathit{inc}\ \mathit{bs}) = 1 + \mathit{toInt}\ \mathit{bs}$. Indeed the cases $\mathit{bs} = \mathit{BNil}$ and $\mathit{bs} = \mathit{B0}\ x$ are immediate, while for the case $\mathit{bs} = \mathit{B1}\ x$, we get

$$\begin{aligned} \mathit{toInt}\ (\mathit{inc}\ (\mathit{B1}\ x)) &= \mathit{toInt}\ (\mathit{B0}\ (\mathit{inc}\ x)) \\ &= 2 * \mathit{toInt}\ (\mathit{inc}\ x) \\ &= 2 * (1 + \mathit{toInt}\ x) \\ &= 2 + 2 * \mathit{toInt}\ x \\ &= 1 + \mathit{toInt}\ (\mathit{B1}\ x) \end{aligned}$$

by unrolling the definitions of *inc* and *toInt* and applying the IH and a bit of algebra. (Although the question did not ask us to prove this property of *inc*, proving it helps us ensure that our function is correct!) □

Question 1.2. This representation contains redundancies, since many different values of type *Bits* can represent the value 0. Write a function $\mathit{nf} :: \mathit{Bits} \to \mathit{Bits}$ that eliminates such redundancies by converting a bit string to *normal form*. It should satisfy the property that $\mathit{toInt}\ (\mathit{nf}\ x) = \mathit{toInt}\ x$, and that $\mathit{toInt}\ (\mathit{nf}\ x) = \mathit{toInt}\ (\mathit{nf}\ y)$ iff $\mathit{nf}\ x = \mathit{nf}\ y$.

Solution : Redundancies arise because *BNil*, *B0 BNil*, *B0* (*B0 BNil*), etc., all represent the number 0. To bring the bit string to normal form, we can recursively inspect and reconstruct it, making sure never to introduce a value of the form *B0 BNil*.

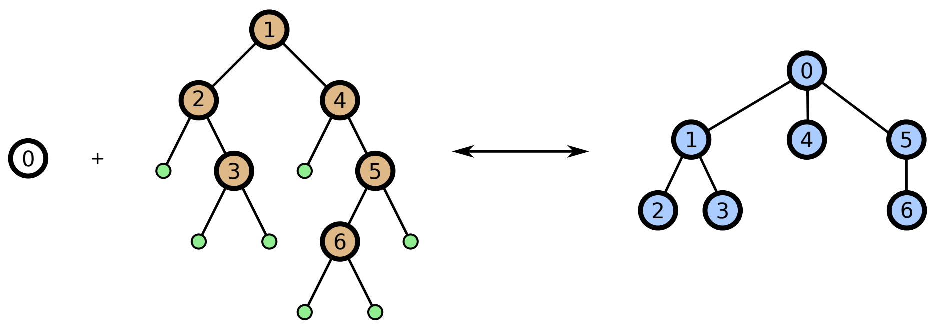
$$
\begin{array}{l}
\mathit{nf} :: \mathit{Bits} \to \mathit{Bits} \\
\mathit{nf}\ \mathit{BNil} = \mathit{BNil} \\
\mathit{nf}\ (\mathit{B0}\ n) = \mathbf{case}\ \mathit{nf}\ n\ \mathbf{of} \\
\quad \mathit{BNil} \to \mathit{BNil} \\
\quad x \to \mathit{B0}\ x \\
\mathit{nf}\ (\mathit{B1}\ n) = \mathit{B1}\ (\mathit{nf}\ n)
\end{array}
$$

Again we can easily verify by induction that $\mathit{toInt}\ (\mathit{nf}\ x) = \mathit{toInt}\ x$, and that $\mathit{toInt}\ (\mathit{nf}\ x) = 0$ just in case $\mathit{nf}\ x = \mathit{BNil}$, from which we can derive that $\mathit{toInt} \circ \mathit{nf}$ is injective. $\square$

For the next question, consider the following data types of binary trees and of general (arbitrary branching) trees with labelled nodes:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{BinTree}\ a = L \mid B\ a\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a) \\
\mathbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a]
\end{array}
$$

The "left-child, right-sibling" encoding defines a correspondence between binary trees and general trees, by interpreting the left child of a binary node as a child and the right child as a sibling in the corresponding general tree. Formally, this interprets a binary tree as a *forest* of general trees, but one can turn a forest into a tree by adding a distinguished root label. We thus have a 1-1 correspondence between pairs (label, binary tree) and general trees:



Question 1.3. Write a pair of functions

$$
\begin{array}{l}
\mathit{toTree} :: (a, \mathit{BinTree}\ a) \to \mathit{Tree}\ a \\
\mathit{fromTree} :: \mathit{Tree}\ a \to (a, \mathit{BinTree}\ a)
\end{array}
$$

implementing the left-child, right-sibling correspondence, and prove that they define a type isomorphism $(a, \mathit{BinTree}\ a) \cong \mathit{Tree}\ a$.

Solution : We find it convenient to write toTree and $\mathit{fromTree}$ in mutual recursion with a pair of functions

$$
\begin{array}{l}
\mathit{toForest} :: \mathit{BinTree}\ a \to [\mathit{Tree}\ a] \\
\mathit{fromForest} :: [\mathit{Tree}\ a] \to \mathit{BinTree}\ a
\end{array}
$$

converting between binary trees and forests:

$$
\begin{array}{l}
\mathit{toTree}\ (x, t) = \mathit{Node}\ x\ (\mathit{toForest}\ t) \\
\mathit{toForest}\ L = [\,] \\
\mathit{toForest}\ (B\ x\ t\ u) = \mathit{toTree}\ (x, t) : \mathit{toForest}\ u \\
\mathit{fromTree}\ (\mathit{Node}\ x\ \mathit{ts}) = (x, \mathit{fromForest}\ \mathit{ts}) \\
\mathit{fromForest}\ [\,] = L \\
\mathit{fromForest}\ (t : \mathit{ts}) = \mathbf{let}\ (x, t') = \mathit{fromTree}\ t\ \mathbf{in}\ B\ x\ t'\ (\mathit{fromForest}\ \mathit{ts})
\end{array}
$$

Likewise, to prove that toTree and $\mathit{fromTree}$ define an isomorphism $(a, \mathit{BinTree}\ a) \cong \mathit{Tree}\ a$, we simultaneously prove by structural induction that $\mathit{toForest}$ and $\mathit{fromForest}$ define an isomorphism $\mathit{BinTree}\ a \cong [\mathit{Tree}\ a]$. The proof is a long but mechanical calculation:

$$
\begin{aligned}
\mathit{fromTree}\ (\mathit{toTree}\ (x,t)) &= \mathit{fromTree}\ (\mathit{Node}\ x\ (\mathit{toForest}\ t)) \\
&= (x, \mathit{fromTree}\ (\mathit{toForest}\ t)) \\
&= (x,t) \\
\mathit{fromForest}\ (\mathit{toForest}\ L) &= \mathit{fromForest}\ [\,] \\
&= L \\
\mathit{fromForest}\ (\mathit{toForest}\ (B\ x\ t\ u)) &= \mathit{fromForest}\ (\mathit{toTree}\ (x,t) : \mathit{toForest}\ u) \\
&= \mathbf{let}\ (x',t') = \mathit{fromTree}\ (\mathit{toTree}\ (x,t))\ \mathbf{in}\ B\ x'\ t'\ (\mathit{fromForest}\ (\mathit{toForest}\ u)) \\
&= \mathbf{let}\ (x',t') = (x,t)\ \mathbf{in}\ B\ x'\ t'\ u \\
&= B\ x\ t\ u \\
\mathit{toTree}\ (\mathit{fromTree}\ (\mathit{Node}\ x\ \mathit{ts})) &= \mathit{toTree}\ (x, \mathit{fromForest}\ \mathit{ts}) \\
&= \mathit{Node}\ x\ (\mathit{toForest}\ (\mathit{fromForest}\ \mathit{ts})) \\
&= \mathit{Node}\ x\ \mathit{ts} \\
\mathit{toForest}\ (\mathit{fromForest}\ [\,]) &= \mathit{toForest}\ L \\
&= [\,] \\
\mathit{toForest}\ (\mathit{fromForest}\ (t : \mathit{ts})) &= \mathit{toForest}\ (\mathbf{let}\ (x,t') = \mathit{fromTree}\ t\ \mathbf{in}\ B\ x\ t'\ (\mathit{fromForest}\ \mathit{ts})) \\
&= \mathbf{let}\ (x,t') = \mathit{fromTree}\ t\ \mathbf{in}\ \mathit{toForest}\ (B\ x\ t'\ (\mathit{fromForest}\ \mathit{ts})) \\
&= \mathbf{let}\ (x,t') = \mathit{fromTree}\ t\ \mathbf{in}\ \mathit{toTree}\ (x,t') : \mathit{toForest}\ (\mathit{fromForest}\ \mathit{ts}) \\
&= \mathit{toTree}\ (\mathit{fromTree}\ t) : \mathit{toForest}\ (\mathit{fromForest}\ \mathit{ts}) \\
&= t : \mathit{ts}
\end{aligned}
$$

□

2 Higher-order functions

Question 2.1. Implement the standard library function $\mathit{maybe} :: b \to (a \to b) \to \mathit{Maybe}\ a \to b$, which internalizes the principle of case-analysis on a value of Maybe type.

Solution :

$$
\begin{aligned}
&\mathit{maybe}\ x\ f\ \mathit{Nothing} = x \\
&\mathit{maybe}\ x\ f\ (\mathit{Just}\ y) = f\ y
\end{aligned}
$$

□

Question 2.2. The standard library function $\mathit{lookup} :: \mathit{Eq}\ a \Rightarrow a \to [(a,b)] \to \mathit{Maybe}\ b$ finds the first value matching a given key in a list of (key, value) pairs. Express lookup in terms of the function $\mathit{find} :: (c \to \mathit{Bool}) \to [c] \to \mathit{Maybe}\ c$, which finds the first value satisfying a predicate in a list, and maybe from the previous question.

Solution :

$$\mathit{lookup}\ x = \mathit{maybe}\ \mathit{Nothing}\ (\mathit{Just} \circ \mathit{snd}) \circ \mathit{find}\ ((\equiv x) \circ \mathit{fst})$$

□

Question 2.3. Express *find* in terms of *foldr*.

Solution :

$$\mathit{find}\ p = \mathit{foldr}\ (\backslash x\ mx \to \textbf{if}\ p\ x\ \textbf{then}\ \mathit{Just}\ x\ \textbf{else}\ mx)\ \mathit{Nothing}$$

□

Question 2.4. Prove the *fusion law* for *foldr*, which states that if h is a function such that $h\ e = e'$ and $h\ (f\ x\ y) = f'\ x\ (h\ y)$, then $h \circ \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e'$.

Solution : Recall the definition of *foldr*:

$$\begin{aligned}&\mathit{foldr}\ f\ e\ [\,] = e\\ &\mathit{foldr}\ f\ e\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)\end{aligned}$$

We prove that $h\ (\mathit{foldr}\ f\ e\ xs) = \mathit{foldr}\ f'\ e'\ xs$ for all xs, by induction on xs:

- (Case $xs = [\,]$): then $h\ (\mathit{foldr}\ f\ e\ [\,]) = h\ e = e' = \mathit{foldr}\ f'\ e'\ [\,]$.
- (Case $xs = x{:}ys$): then $h\ (\mathit{foldr}\ f\ e\ (x{:}ys)) = h\ (f\ x\ (\mathit{foldr}\ f\ e\ ys)) = f'\ x\ (h\ (\mathit{foldr}\ f\ e\ ys)) = f'\ x\ (\mathit{foldr}\ f'\ e'\ ys) = \mathit{foldr}\ f'\ e'\ (x : ys)$.

□

Question 2.5. Using your answers to the previous questions, *derive* an expression for *lookup* using only *foldr*. (For full credit, you must justify the expression using the fusion law.)

Solution : By Q2.2 and Q2.3, we have that $\mathit{lookup}\ x = h \circ \mathit{foldr}\ f\ e$ where

$$\begin{aligned}f &= \backslash kv\ mx \to \textbf{if}\ \mathit{fst}\ kv \equiv x\ \textbf{then}\ \mathit{Just}\ kv\ \textbf{else}\ mx\\ e &= \mathit{Nothing}\\ h &= \mathit{maybe}\ \mathit{Nothing}\ (\mathit{Just} \circ \mathit{snd})\end{aligned}$$

We can calculate that

$$\begin{aligned}h\ (f\ kv\ mx) &= \textbf{if}\ \mathit{fst}\ kv \equiv x\ \textbf{then}\ \mathit{Just}\ (\mathit{snd}\ kv)\ \textbf{else}\ h\ mx\\ h\ e &= \mathit{Nothing}\end{aligned}$$

and hence we can conclude that

$$\mathit{lookup}\ x = \mathit{foldr}\ (\backslash kv\ mx \to \textbf{if}\ \mathit{fst}\ kv \equiv x\ \textbf{then}\ \mathit{Just}\ (\mathit{snd}\ kv)\ \textbf{else}\ mx)\ \mathit{Nothing}$$

by the fusion law. □

3 λ-calculus and propositions-as-types

Question 3.1. For each of the Haskell functions below, state whether the function is typable, and if so give its most general type.

$$\begin{aligned}&\mathit{f1} = \backslash x \to [x, x \circ x, x \circ x \circ x]\\ &\mathit{f2} = \backslash x\ y \to (y, x + y)\\ &\mathit{f3} = \backslash x\ y \to x + y\ x + y\end{aligned}$$

Solution :

$f1 :: (a \to a) \to [a \to a]$

We infer that the argument x of *f1* must have type $a \to a$ since it is composed with itself, and we can verify that $[x, x \circ x, x \circ x \circ x]$ is a valid list of type $[a \to a]$.

$f2 :: \mathit{Num}\ a \Rightarrow a \to a \to (a, a)$

The arguments of *f2* are added together so both must be numbers, and the function returns a pair of numbers.

f3 untypable

The argument y of *f3* must be both a number and a function from numbers to numbers, which is impossible. □

Peirce's law is a propositional tautology stating that $((P \supset Q) \supset P) \supset P$. Although true classically, Peirce's law is not constructively valid, and there is no simply-typed lambda term whose principal type is $((p \to q) \to p) \to p$.

Question 3.2. Assume that you are given a polymorphic function $\mathit{peirce} :: ((p \to q) \to p) \to p$ in Haskell. Show how you could use it to define a polymorphic expression $\mathit{lem} :: \mathit{Either}\ p\ (p \to \mathit{Void})$ realizing the law of excluded middle. (Here *Void* is the data type with no constructors.)

Solution : Let's begin by describing a constructive proof in natural language. Assuming Peirce's law, we have in particular (by letting P be $P \vee \neg P$ and Q be $\bot$) that

$$(((P \vee \neg P) \supset \bot) \supset (P \vee \neg P)) \supset (P \vee \neg P) \tag{1}$$

and so to derive the law of excluded middle $P \vee \neg P$ it suffices to prove that

$$((P \vee \neg P) \supset \bot) \supset (P \vee \neg P) \tag{2}$$

and apply modus ponens to (1) and (2). To show (2), we assume

$$(P \vee \neg P) \supset \bot \tag{3}$$

and derive $P \vee \neg P$, for which it suffices to show $\neg P$. So assume by way of contradiction that

$$P \tag{4}$$

is true. Then $P \vee \neg P$ is true. But that contradicts the hypothesis (3). QED.

The argument above translates directly to the following Haskell code, where hypothesis (3) corresponds to the variable f, and (4) to the variable x:

$\mathit{lem} = \mathit{peirce}\ (\backslash f \to \mathit{Right}\ (\backslash x \to f\ (\mathit{Left}\ x)))$

□

4 Side-effects and monads

For the next few questions, consider the type

$$\textbf{data}\ \mathit{Logged}\ a = \mathit{Logged}\ a\ [\mathit{String}]$$

whose values consist of values of type a paired with a list of strings. We can think of values of type $\mathit{Logged}\ a$ as computations producing a sequence of string outputs on the way to a value.

Question 4.1. Show Logged is a functor, by defining $\mathit{fmap} :: (a \to b) \to \mathit{Logged}\ a \to \mathit{Logged}\ b$ and verifying that $\mathit{fmap}\ \mathit{id} = \mathit{id}$ and $\mathit{fmap}\ (f \circ g) = \mathit{fmap}\ f \circ \mathit{fmap}\ g$ for all f and g.

Solution :

$$\mathit{fmap}\ f\ (L\ x\ l) = L\ (f\ x)\ l$$

The functor laws are verified immediately. □

Question 4.2. Turn Logged into a monad by defining

$$\begin{aligned}&\mathit{return} :: a \to \mathit{Logged}\ a\\ &(\gg\!=) :: \mathit{Logged}\ a \to (a \to \mathit{Logged}\ b) \to \mathit{Logged}\ b\end{aligned}$$

so that return creates a pure computation with an empty log, while $(\gg\!=)$ runs a logged computation and feeds the result to a log-producing continuation, concatenating the two logs. Then, prove that these definitions satisfy the three monad laws

$$(\mathit{return}\ x \gg\!= f) = f\ x \quad (\mathit{lx} \gg\!= \mathit{return}) = \mathit{lx} \quad (\mathit{lx} \gg\!= f) \gg\!= g = \mathit{lx} \gg\!= (\backslash x \to f\ x \gg\!= g)$$

for all $x :: a$, $\mathit{lx} :: \mathit{Logged}\ a$, $f :: a \to \mathit{Logged}\ b$, and $g :: b \to \mathit{Logged}\ c$.

Solution :

$$\begin{aligned}&\mathit{return}\ x = \mathit{Logged}\ x\ [\,]\\ &\mathit{Logged}\ x\ l1 \gg\!= f = \textbf{let}\ \mathit{Logged}\ y\ l2 = f\ x\ \textbf{in}\ \mathit{Logged}\ y\ (l1 +\!\!+ l2)\end{aligned}$$

We verify the monad laws:

$$\begin{aligned}
\mathit{return}\ x \gg\!= f &= L\ x\ [\,] \gg\!= f\\
&= \textbf{let}\ \mathit{Logged}\ y\ l = f\ x\ \textbf{in}\ \mathit{Logged}\ y\ ([\,] +\!\!+ l)\\
&= f\ x\\
(\mathit{Logged}\ x\ l1 \gg\!= \mathit{return}) &= \textbf{let}\ \mathit{Logged}\ y\ l2 = \mathit{Logged}\ x\ [\,]\ \textbf{in}\ \mathit{Logged}\ y\ (l1 +\!\!+ l2)\\
&= \mathit{Logged}\ x\ (l1 +\!\!+ [\,])\\
&= \mathit{Logged}\ x\ l1\\
(\mathit{Logged}\ x\ l1 \gg\!= f) \gg\!= g &= (\textbf{let}\ \mathit{Logged}\ y\ l2 = f\ x\ \textbf{in}\ \mathit{Logged}\ y\ (l1 +\!\!+ l2)) \gg\!= g\\
&= \textbf{let}\ \mathit{Logged}\ y\ l2 = f\ x\ \textbf{in}\ (\mathit{Logged}\ y\ (l1 +\!\!+ l2) \gg\!= g)\\
&= \textbf{let}\ \mathit{Logged}\ y\ l2 = f\ x\ \textbf{in}\ \textbf{let}\ \mathit{Logged}\ z\ l3 = g\ y\ \textbf{in}\ \mathit{Logged}\ z\ ((l1 +\!\!+ l2) +\!\!+ l3)\\
&= \textbf{let}\ \mathit{Logged}\ y\ l2 = f\ x\ \textbf{in}\ \textbf{let}\ \mathit{Logged}\ z\ l3 = g\ y\ \textbf{in}\ \mathit{Logged}\ z\ (l1 +\!\!+ (l2 +\!\!+ l3))\\
&= \mathit{Logged}\ x\ l1 \gg\!= (\backslash x \to f\ x \gg\!= g)
\end{aligned}$$

□

Question 4.3. Define a function $\mathit{log} :: \mathit{String} \to \mathit{Logged}\ ()$ representing a computation that writes a single string to the log and returns a trivial value.

Solution :

$$\mathit{log}\ s = L\ ()\ [s]$$

□

Question 4.4. Consider the following data type of arithmetic expressions:

$$\mathbf{data}\ \mathit{Exp} = \mathit{Con}\ \mathit{Double} \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Mul}\ \mathit{Exp}\ \mathit{Exp}$$

Define a function $\mathit{evalLogged} :: \mathit{Exp} \to \mathit{Logged}\ \mathit{Double}$ that evaluates an expression to a number together with a log recording the order in which the subexpressions are evaluated.

Solution :

```
evalLogged e = do
  log ("evaluating: " ++ show e)
  case e of
    Con x -> return x
    Add e1 e2 -> do
      x1 <- evalLogged e1
      x2 <- evalLogged e2
      return (x1 + x2)
    Mul e1 e2 -> do
      x1 <- evalLogged e1
      x2 <- evalLogged e2
      return (x1 * x2)
```

□

5 Laziness and infinite objects

The goal of this last section is to write some *seemingly impossible functional programs.*[1] The *Cantor space* 2^ω is a topological space whose elements are infinite sequences of binary digits. To formalize the Cantor space in Haskell, let's begin by recalling the (co)data type *Stream a* of infinite sequences of *a*s:

$$\mathbf{data}\ \mathit{Stream}\ a = \mathit{Stream}\ \{\mathit{hd} :: a, \mathit{tl} :: \mathit{Stream}\ a\}$$

We then define the Cantor space as the type of streams of bits:

```
data Bit = Zero | One deriving (Eq)
type Cantor = Stream Bit
```

Question 5.1. Define an operation

$$(\#) :: \mathit{Bit} \to \mathit{Cantor} \to \mathit{Cantor}$$

that prepends a bit to the beginning of an infinite sequence of bits.

[1] Adapted from Martín Escardó's guest article of the same name on the *Mathematics and Computation* blog (Escardó, September 2007).

Solution :

$$b \;\#\; bs = \mathit{Stream}\;\{\mathit{hd} = b, \mathit{tl} = bs\}$$

□

To gain some more intuition, let's show that values of type *Cantor* may be interpreted as functions from natural numbers to bits, and vice versa. (For convenience, we'll write *Nat* as a type synonym for *Int*, but with the understanding that its values are restricted to non-negative integers.)

Question 5.2. Implement a pair of coercions $\mathit{fromCantor} :: \mathit{Cantor} \to (\mathit{Nat} \to \mathit{Bit})$ and $\mathit{toCantor} :: (\mathit{Nat} \to \mathit{Bit}) \to \mathit{Cantor}$. Your functions should realize a type isomorphism $\mathit{Cantor} \cong \mathit{Nat} \to \mathit{Bit}$, although you do not need to prove this fact.

Solution :

$$\begin{array}{l}
\mathit{fromCantor}\; s\; n \\
\quad \mid n \equiv 0 = \mathit{hd}\; s \\
\quad \mid n > 0 = \mathit{fromCantor}\; (\mathit{tl}\; s)\; (n-1) \\
\mathit{toCantor}\; f = \mathit{Stream}\;\{\mathit{hd} = f\;0, \mathit{tl} = \mathit{toCantor}\;(\backslash n \to f\;(n+1))\}
\end{array}$$

□

Now, our goal will be to write a higher-order function

$$\mathit{existsC} :: (\mathit{Cantor} \to \mathit{Bool}) \to \mathit{Bool}$$

which takes a total predicate over the Cantor space, and decides if there is some infinite sequence of bits making the predicate true. By *total* predicate, we mean a function $\mathit{Cantor} \to \mathit{Bool}$ that terminates for all input values. For example,

$$\mathit{good}\; s = \mathit{fromCantor}\; s\; 3 \not\equiv \mathit{fromCantor}\; s\; 4 \;\&\&\; \mathit{fromCantor}\; s\; 7 \equiv \mathit{fromCantor}\; s\; 15$$

is a total predicate (testing that bits 3 and 4 are distinct, and bits 7 and 15 are equal), but

$$\mathit{bad}\; s = \mathit{hd}\; s \equiv \mathit{One} \;\&\&\; \mathit{bad}\;(\mathit{tl}\; s)$$

is not total, since it will not terminate on an infinite sequence of *One*s.

Question 5.3. Write *existsC* in mutual recursion with a function

$$\mathit{findC} :: (\mathit{Cantor} \to \mathit{Bool}) \to \mathit{Cantor}$$

so that they satisfy the following specifications:

- For any total predicate p, $\mathit{existsC}\; p$ should terminate and return *True* just in case there exists a sequence $bs :: \mathit{Cantor}$ such that $p\; bs = \mathit{True}$, or else return *False*.
- For any total predicate p, $\mathit{findC}\; p$ should terminate and return a sequence $bs :: \mathit{Cantor}$ such that $p\; bs = \mathit{True}$ if there exists such a sequence, or else return an arbitrary sequence.

Hint: Put your faith in the specifications to define these functions by mutual recursion. Keep in mind that findC needs to return an infinite sequence of bits, so in particular it needs to determine the initial bit.

Solution :

$$\begin{array}{l} \mathit{existsC}\ p = p\ (\mathit{findC}\ p) \\ \mathit{findC}\ p = \textbf{if}\ \mathit{existsC}\ (\backslash bs \rightarrow p\ (\mathit{Zero} \# bs))\ \textbf{then}\ \mathit{Zero} \# \mathit{findC}\ (\backslash bs \rightarrow p\ (\mathit{Zero} \# bs)) \\ \quad \textbf{else}\ \mathit{One} \# \mathit{findC}\ (\backslash bs \rightarrow p\ (\mathit{One} \# bs)) \end{array}$$

(For a deeper discussion of these definitions, see Escardó's article mentioned above.) □

Question 5.4. Define an operation

$$\mathit{equalC} :: \mathit{Eq}\ a \Rightarrow (\mathit{Cantor} \rightarrow a) \rightarrow (\mathit{Cantor} \rightarrow a) \rightarrow \mathit{Bool}$$

that takes two total functions over the Cantor space into some type admitting decidable equality, and decides whether the functions are equal on all inputs. You can assume as given *existsC* and *findC* satisfying the specifications from the previous question.

Solution :

$$\mathit{equalC}\ f\ g = \mathit{not}\ (\mathit{existsC}\ (\backslash c \rightarrow f\ c \not\equiv g\ c))$$

To check that two functions are equal, we test that they do not disagree on any input sequence. (Remarkably, *equalC f g* is well-defined and always terminates, even though it looks like there are infinitely many, infinitely long sequences to test!) □