CSE301 / CSC_3F001_EP Functional Programming, Fall 2024

In-class exam

(solutions)

This exam consists of five parts that can be solved in any order. For questions that ask you to write some kind of function, you should assume that the input satisfies whatever conditions are given in the specification of the function. For example, if a question asks you to write a function $sqrt :: Double \to Double$ computing the square root of a non-negative floating point number, you do not need to handle the case that the input is negative.

1 First-order data types

Consider the following data type of (positive) boolean formulas:

type $Id = Int$
data $Frm = Atm\ Id \mid And\ Frm\ Frm \mid Or\ Frm\ Frm$

In words, a formula is either an atomic formula carrying an (integer) identifier, or the conjunction or disjunction of two boolean formulas.

Question 1.1. What is the type of the constructor And?

Solution : $And :: Frm \to Frm \to Frm$ □

Question 1.2. Write a function $isAtm :: Frm \to Bool$ that tests whether a formula is atomic.

Solution :

$isAtm\ (Atm\ \_) = True$
$isAtm\ \_ = False$

□

The truth or falsity of a boolean formula is determined by the truth or falsity of its atoms. For example, the formula $And\ (Atm\ 1)\ (Or\ (Atm\ 2)\ (Atm\ 3))$ is true just in case atoms 1 and 2 are true or atoms 1 and 3 are true.

Question 1.3. Write a function $eval :: Frm \to [Id] \to Bool$ that evaluates a boolean formula to either $True$ or $False$, given a list of all the identifiers of true atoms. For example, letting $p = And\ (Atm\ 1)\ (Or\ (Atm\ 2)\ (Atm\ 3))$ as above, you should have $eval\ p\ [1,2] = True$ and $eval\ p\ [1,3] = True$ but $eval\ p\ [2,3] = False$.

Solution : We make use of the standard library function $elem :: Eq\ a \Rightarrow a \to [a] \to Bool$ to test whether a given atomic formula identifier is in the list of true identifiers, and interpret And and Or using (&&) and (||) respectively:

$eval\ (Atm\ x)\ rho = elem\ x\ rho$
$eval\ (And\ p\ q)\ rho = eval\ p\ rho\ \&\&\ eval\ q\ rho$
$eval\ (Or\ p\ q)\ rho = eval\ p\ rho\ ||\ eval\ q\ rho$

□

A formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of one or more disjunctions of one or more atomic formulas. For example, $And\ (Atm\ 1)\ (Or\ (Atm\ 2)\ (Atm\ 3))$ is in CNF, but $Or\ (And\ (Atm\ 1)\ (Atm\ 2))\ (And\ (Atm\ 1)\ (Atm\ 3))$ is not.

Question 1.4. Write a function $isCNF :: Frm \to Bool$ that tests whether a formula is in conjunctive normal form.

Solution : It is convenient to first introduce a helper function *isClause* that tests whether a formula is a big disjunction of one or more atoms, and then write *isCNF* to test that a formula is a big conjunction of clauses:

$$
\begin{aligned}
&isClause :: Frm \to Bool \\
&isClause\ (Atm\ \_) = True \\
&isClause\ (Or\ p\ q) = isClause\ p\ \&\&\ isClause\ q \\
&isClause\ (And\ \_\ \_) = False \\
&isCNF :: Frm \to Bool \\
&isCNF\ (And\ p\ q) = isCNF\ p\ \&\&\ isCNF\ q \\
&isCNF\ p = isClause\ p
\end{aligned}
$$

□

2 Higher-order functions

The first three problems in this section ask you to implement various transformations on lists of numbers. To get full credit, you must implement them using only higher-order functions or list comprehensions, without using recursion.

Question 2.1. Write a function $evenSucc :: [Int] \to [Int]$ that takes a list of numbers and returns the successors of all of the even numbers in the list. (Example: $evenSucc\ [1, 5, 8, 12, 15] = [9, 13]$.)

Solution : Using *filter* and *map*:

$$evenSucc = map\ (+1) \circ filter\ (\backslash x \to x\ `mod`\ 2 \equiv 0)$$

Using *foldr*:

$$evenSucc = foldr\ (\backslash x\ ys \to \textbf{if}\ x\ `mod`\ 2 \equiv 0\ \textbf{then}\ (x + 1) : ys\ \textbf{else}\ ys)\ [\,]$$

Using a list comprehension:

$$evenSucc\ xs = [1 + x \mid x \leftarrow xs, x\ `mod`\ 2 \equiv 0]$$

□

Question 2.2. Write a function $diffs :: [Int] \to [Int]$ that takes a non-empty list of numbers $[x_0, \ldots, x_n]$ and returns the list of differences $[x_1 - x_0, x_2 - x_1, \ldots, x_n - x_{n-1}]$. (Example: $diffs\ [3, 1, 4, 1, 5, 9] = [-2, 3, -3, 4, 4]$.)

Solution : Using *zipWith*:

$$diffs\ xs = zipWith\ (-)\ (tail\ xs)\ xs$$

□

Question 2.3. Write a function $decimal :: [Int] \to Int$ that takes a list of digits between 0 and 9 and interprets it as the decimal encoding of a number. (Example: $decimal\ [3, 1, 4, 1, 5, 9] = 314159$.)

Solution : Using *foldl*:

$$decimal = foldl\ (\backslash n\ d \to n * 10 + d)\ 0$$

Using *foldr*:

$$decimal = snd \circ foldr\ (\backslash d\ np \to (10 * fst\ np, d * fst\ np + snd\ np))\ (1, 0)$$

(Observe that the function is a bit easier to define using *foldl* due to the natural left-to-right reading of decimal numbers, e.g., $314159 = (((((0*10+3)*10+1)*10+4)*10+1)*10+5)*10+9$.) □

We return now to the data type *Frm* of boolean formulas introduced in §1, but allowing for the representation of non-positive formulas by interpreting negative identifiers as negated atoms. For example, we now interpret $p = And\ (Atm\ 1)\ (Or\ (Atm\ (-2))\ (Atm\ (-1)))$ as asserting that atom 1 is true and that either atom 2 is false or atom 1 is false. The goal of the next three questions is to write a satisfiability tester in continuation-passing style.

A formula is said to be *satisfiable* if there is some assignment of truth values to atoms making the whole formula true. For example, the formula p above is satisfiable by the assignment $(1 \mapsto True,\ 2 \mapsto False)$, but the formula $And\ p\ (Atm\ 2)$ is not satisfiable. The same assignment $(1 \mapsto True,\ 2 \mapsto False)$ may also be considered as satisfying the formula $Or\ p\ (Atm\ 3)$ even though it does not assign a value to the atom 3, which can be arbitrary. To make this more precise let us introduce a type of *partial assignments*.

$$\textbf{type}\ PAsn = Id \to Maybe\ Bool$$

A partial assignment $\sigma :: PAsn$ extends another partial assignment $\rho :: PAsn$, written $\rho \leq \sigma$, if for all $x :: Id$, either $\rho(x) = Nothing$, or $\rho(x) = Just\ b$ and $\sigma(x) = Just\ b$. We say that a partial assignment ρ satisfies a formula p if the latter evaluates to *True* under any extension of ρ to a complete assignment $\sigma \geq \rho$ for all the variables in p.

Question 2.4. Define a partial assignment $\bot$ that is below every other partial assignment $\bot \leq \rho$.

Solution : We take the constant function that always returns *Nothing*, i.e., $\bot = const\ Nothing$. □

Let $k :: PAsn \to Bool$ be a predicate on partial assignments. We say that k is *monotonic* if $k\ \rho = True$ and $\rho \leq \sigma$ implies $k\ \sigma = True$ for all ρ and σ.[1]

Question 2.5. Write a higher-order function

$$sat :: Frm \to PAsn \to (PAsn \to Bool) \to Bool$$

that takes as input a formula p, a partial assignment ρ, and a ~~monotonic~~ **anti-monotonic** predicate k, and returns *True* just in case there is some partial assignment $\sigma \geq \rho$ satisfying p such that $k\ \sigma = True$.

Solution : *sat* is written in continuation-passing style, similar to the definition of the regular expression matcher *acc* from Lab 2:

```
sat (Atm x) rho k = case rho (abs x) of
  Just b → if b ≡ (x > 0) then k rho else False
  Nothing → k (\y → if y ≡ abs x then Just (x > 0) else rho y)
```

[1] **Correction 23/10/2024:** the specification of *sat* in Q2.5 should rather have asked that the predicate k be *anti-monotonic* in the sense that $k\ \sigma = True$ and $\rho \leq \sigma$ implies $k\ \rho = True$. I thank Timofey Fedoseev for alerting me to this issue.

$sat\ (Or\ p\ q)\ rho\ k = sat\ p\ rho\ k\ ||\ sat\ q\ rho\ k$
$sat\ (And\ p\ q)\ rho\ k = sat\ p\ rho\ (\backslash rho' \to sat\ q\ rho'\ k)$

In the atomic case, we check whether the underlying variable $abs\ x$ is already assigned a value in rho. If it is, then the formula $Atm\ x$ is satisfied by any extension $sigma$ of rho just in case the value assigned by rho is equal to the sign of x, and moreover if $k\ sigma = True$ then $k\ rho = True$ by anti-monotonicity. ("**if** $b \equiv (x > 0)$ **then** $k\ rho$ **else** $False$") Otherwise, if $abs\ x$ is not already assigned a value in rho, then we extend it precisely by the assignment that makes the formula $Atm\ x$ true and call the continuation. ("$k\ (\backslash y \to$ **if** $y \equiv abs\ x$ **then** $Just\ (x > 0)$ **else** $rho\ y$)")

In the case of a disjunction we use that a satisfying assignment for $Or\ p\ q$ is either a satisfying assignment for p or a satisfying assignment for q. ("$sat\ p\ rho\ k \mid sat\ q\ rho\ k$")

Finally, it is in the case of conjunction $And\ p\ q$ where we really make use of continuation-passing style. We begin by trying to satisfy the first conjunct, and in the event that there is some extension rho' of rho that satisfies p, then (by appropriately setting the continuation to $sat\ p\ rho$) we try to find an extension of rho' that satisfies q. ("$sat\ p\ rho\ (\backslash rho' \to sat\ q\ rho'\ k)$") Note that the specification of sat ensures that if k is anti-monotonic then so will be the predicate $\backslash rho' \to sat\ q\ rho'\ k$, which is needed in order to prove the correctness of sat by structural induction on the input formula. □

Question 2.6. Write a function $satisfiable :: Frm \to Bool$ that tests if a formula is satisfiable by an appropriate call to sat. (You can assume given a correct implementation of sat.)

Solution : We call $sat\ p\ rho\ k$ with the smallest partial assignment $rho = [\,]$ and the trivial anti-monotonic predicate $k = const\ True$.

$satisfiable\ p = sat\ p\ [\,]\ (const\ True)$

The specification of sat ensures that $sat\ p\ [\,]\ (const\ True)$ returns $True$ iff there is a partial assignment satisfying p, which exists iff p is satisfiable. □

3 λ-calculus and typing

Recall that the *Church numeral* $\overline{n}$ is defined as the lambda term

$$\overline{n} = \lambda f.\lambda x.\ \underbrace{f(\ldots f}_{n \text{ times}}(x)\ldots)$$

and recall the following definitions of the successor and addition operations:

$$suc = \lambda m.\lambda f.\lambda x.f\ (m\ f\ x)$$
$$add = \lambda m.\lambda n.m\ suc\ n$$

For the next two questions let $e = add\ \overline{3}\ \overline{2}$.

Question 3.1. Give an example of a sequence of four β-reduction steps $e \to e_1 \to e_2 \to e_3 \to e_4$ (multiple answers are possible).

Solution : For example, the following sequence of reductions (underlining the β-redices reduced):

$$\begin{aligned} e = \underline{add\ \overline{3}}\ \overline{2} &\to (\lambda n.\overline{3}\ suc\ n)\ \overline{2} = \underline{(\lambda n.\overline{3}\ suc\ n)\ \overline{2}} \\ &\to \overline{3}\ suc\ \overline{2} = \underline{\overline{3}\ suc}\ \overline{2} \\ &\to (\lambda x.suc\ (suc\ (suc\ x)))\ \overline{2} = \underline{(\lambda x.suc\ (suc\ (suc\ x)))\ \overline{2}} \\ &\to suc\ (suc\ (suc\ \overline{2})) \end{aligned}$$

Alternatively we could keep the first and last steps as above but perform

$$(\lambda n.\overline{3}\ suc\ n)\ \overline{2} = (\lambda n.\underline{\overline{3}\ suc}\ n)\ \overline{2} \to (\lambda n.(\lambda x.suc\ (suc\ suc\ x))\ n)\ \overline{2} = \underline{(\lambda n.(\lambda x.suc\ (suc\ (suc\ x)))\ n)\ \overline{2}}$$
$$\to (\lambda n.suc\ (suc\ (suc\ n)))\ \overline{2}$$

as intermediate steps of β-reduction. (Many other reduction sequences are possible.) □

Question 3.2. What is the β-normal form of e?

Solution : $\lambda f.\lambda x.f(f(f(f(f(x))))) = \overline{5}$. □

For the next two questions we consider lambda expressions in Haskell syntax.

Question 3.3. What is the principal type of $aba = \backslash a\ b\ x \to a\ (b\ (a\ x))$?

Solution : $aba :: (t1 \to t2) \to (t2 \to t1) \to t1 \to t2$.

Since x is only passed as an argument to a it has a completely unconstrained type $t1$, but the fact that the output of a (respectively b) is passed as an input to b (respectively a) forces a and b to have function types $t1 \to t2$ and $t2 \to t1$, respectively, for an arbitrary $t2$. □

Question 3.4. What is the principal type of $abba = \backslash a\ b\ x \to a\ (b\ (b\ (a\ x)))$?

Solution : $abba :: (t \to t) \to (t \to t) \to t \to t$.

The reasoning is as for the previous question, but the fact that the output of b is also passed as an input to b imposes the additional equation $t1 = t2$. □

4 Side-effects and monads

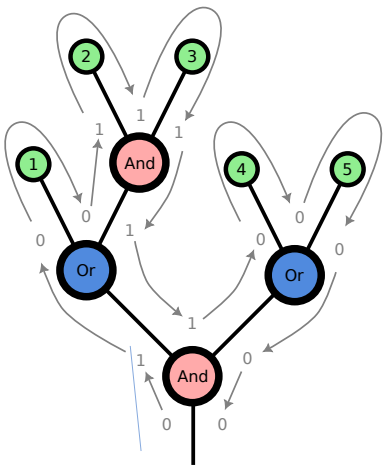
We again refer to the data type of boolean formulas *Frm* defined in §1. We say that a boolean formula is *andor-bracketed* if it has the same number of conjunctions as disjunctions, and moreover the number of conjunctions always upper bounds the number of disjunctions in a left-to-right traversal of the formula. For example, the formula

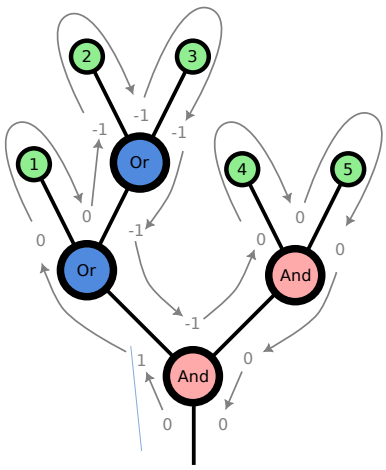
$$And\ (Or\ (Atm\ 1)\ (And\ (Atm\ 2)\ (Atm\ 3)))\ (Or\ (Atm\ 4)\ (Atm\ 5))$$

is andor-bracketed, but the formula

$$And\ (Or\ (Atm\ 1)\ (Or\ (Atm\ 2)\ (Atm\ 3)))\ (And\ (Atm\ 4)\ (Atm\ 5))$$

is not. We can verify this by walking along the formula trees from left to right while keeping track of the difference between the number of *And*s and *Or*s that we've already seen:





We refer to the difference between the number of *And*s and *Or*s already seen at any given point on the walk as the *excedance* at that point. A formula is andor-bracketed just in case its excedance never goes below 0.

Question 4.1. Write a function $walk :: Frm \to Int \to Maybe\ Int$ that takes as input a formula p and a non-negative integer n, and returns the value $Just\ (n + a - o)$, where a is the number of *And*s and o is the number of *Or*s in p, just in case the excedance of p never goes below $-n$. Otherwise (i.e., if the excedance of p goes below $-n$ at some point), it should return *Nothing*.

Solution :

$$
\begin{aligned}
&walk :: Frm \to Int \to Maybe\ Int \\
&walk\ (Atm\ \_)\ n = Just\ n \\
&walk\ (And\ p\ q)\ n = walk\ p\ (n+1) \gg\!= \backslash n' \to walk\ q\ n' \\
&walk\ (Or\ p\ q)\ n \\
&\quad \mid n > 0 = walk\ p\ (n-1) \gg\!= \backslash n' \to walk\ q\ n' \\
&\quad \mid otherwise = Nothing
\end{aligned}
$$

This is a direct encoding of state-passing style, using the bind operator of the *Maybe* monad to simplify handling of "exceptions" (i.e., when the excedance counter goes below zero). Observe that the excedance counter is "incremented" when we pass an *And* node and "decremented" when we pass an *Or* node (putting these verbs in scare quotes because the input variable n is never actually modified, just replaced by either $n + 1$ or $n - 1$ when calling *walk p* recursively). □

Question 4.2. Write a function $isAOB :: Frm \to Bool$ that tests if a formula is andor-bracketed using an appropriate call to *walk*. (You can assume given a correct implementation of *walk*.)

Solution : $isAOB\ p = walk\ p\ 0 \equiv Just\ 0$. □

5 Laziness and infinite objects

The standard library function *unfoldr* constructs a potentially infinite list by repeatedly iterating a function to an initial seed, with the possibility of stopping. The definition of *unfoldr* given in the standard library is equivalent to the following:

$$
\begin{aligned}
&unfoldr\ f\ s = \textbf{case}\ f\ s\ \textbf{of} \\
&\quad Nothing \to [\,] \\
&\quad Just\ (a, s') \to a : unfoldr\ f\ s'
\end{aligned}
$$

Question 5.1. What is the principal type of *unfoldr*?

Solution : $unfoldr :: (b \to Maybe\ (a, b)) \to b \to [a]$ □

Question 5.2. Let *Five* be a data type with five constructors:

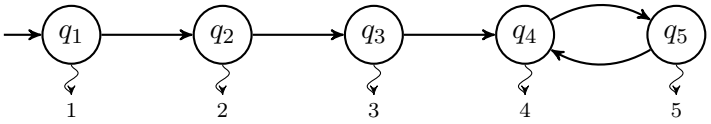
$$\textbf{data}\ Five = Q1 \mid Q2 \mid Q3 \mid Q4 \mid Q5$$

Define a function f such that $unfoldr\ f\ Q1$ builds the list $[1, 2, 3, 4, 5, 4, 5\,..]$ that starts 1, 2, 3 and ends with an infinitely repeating sequence of 4s and 5s.

Solution : Here is one possibility:

$$
\begin{aligned}
&f\ Q1 = \mathit{Just}\ (1, Q2)\\
&f\ Q2 = \mathit{Just}\ (2, Q3)\\
&f\ Q3 = \mathit{Just}\ (3, Q4)\\
&f\ Q4 = \mathit{Just}\ (4, Q5)\\
&f\ Q5 = \mathit{Just}\ (5, Q4)
\end{aligned}
$$

We can think of the values $Q1, \ldots, Q5$ as the states of an automaton, with f encoding the transition function as well as an output value at each state:



Then $\mathit{unfoldr}\ f\ Q1 = [1, 2, 3, 4, 5, 4, 5\,..]$ describes the observed output of the automaton when run starting from the initial state. □

For the next two questions, consider the function *uncons* defined by:

$$
\begin{aligned}
&\mathit{uncons} :: [a] \to \mathit{Maybe}\ (a, [a])\\
&\mathit{uncons}\ [\,] = \mathit{Nothing}\\
&\mathit{uncons}\ (x : xs) = \mathit{Just}\ (x, xs)
\end{aligned}
$$

Question 5.3. Prove that $\mathit{unfoldr}\ \mathit{uncons}\ xs = xs$ for any finite list $xs :: [a]$.

Solution : We prove this by structural induction on xs:

- (Base case $xs = [\,]$.) Since $\mathit{uncons}\ [\,] = \mathit{Nothing}$, the **case** expression in the definition of *unfoldr* simplifies to $\mathit{unfoldr}\ \mathit{uncons}\ [\,] = [\,]$.
- (Inductive case $xs = x : xs'$.) Since $\mathit{uncons}\ (x : xs') = \mathit{Just}\ (x, xs')$, the **case** expression in the definition of *unfoldr* simplifies to $\mathit{unfoldr}\ \mathit{uncons}\ (x : xs') = x : \mathit{unfoldr}\ \mathit{uncons}\ xs'$, which is equal to $x : xs'$ by the induction hypothesis.

(Note that structural induction is only a valid proof principle for finite lists, which is why this argument does not work for infinite lists.) □

Recall the definition of the operation $(!!) :: [a] \to \mathit{Int} \to a$ extracting the nth element of a list:

$$
\begin{aligned}
&[\,] \mathbin{!!} n = \mathit{error}\ \texttt{"!!: index too large"}\\
&(x : xs) \mathbin{!!} n\\
&\ \mid n \equiv 0 = x\\
&\ \mid n > 0 = xs \mathbin{!!} (n - 1)
\end{aligned}
$$

We say that two potentially infinite (that is, either finite or infinite) lists xs and ys are *observationally equivalent*, written $xs \approx ys$, if for all n, either $xs \mathbin{!!} n = ys \mathbin{!!} n$ or both $xs \mathbin{!!} n$ and $ys \mathbin{!!} n$ raise an error. It is easy to check that the observational equivalence relation is reflexive, symmetric, and transitive.

Question 5.4. Prove that $\mathit{unfoldr}\ \mathit{uncons}\ xs \approx xs$ for any finite or infinite list xs. (You can take for given the result of the previous question.)

Solution : If xs is finite then $\mathit{unfoldr}\ \mathit{uncons}\ xs = xs$ by the previous question, and hence $\mathit{unfoldr}\ \mathit{uncons}\ xs \approx xs$ by reflexivity of observational equivalence. Let us therefore assume that xs is infinite, and establish that $(\mathit{unfoldr}\ \mathit{uncons}\ xs) \mathbin{!!} n = xs \mathbin{!!} n$ for all n. The proof is by induction on n.

- (Base case $n = 0$.) Since xs is infinite, we know that it must be of the form $xs = x : xs'$ for some x and xs'. Then, since $uncons\ (x : xs') = Just\ (x, xs')$, the **case** expression in the definition of $unfoldr$ simplifies to $unfoldr\ uncons\ (x : xs') = x : unfoldr\ uncons\ xs'$, and we can derive that

$$\begin{aligned}
(unfoldr\ uncons\ xs) \mathbin{!!} n &= (x : unfoldr\ uncons\ xs') \mathbin{!!} 0 \\
&= x && \text{(definition of (!!))} \\
&= (x : xs') \mathbin{!!} 0 \\
&= xs \mathbin{!!} n.
\end{aligned}$$

- (Inductive case $n > 0$.) Again we know that xs is of the form $xs = x : xs'$, and use the equation $uncons\ (x : xs') = Just\ (x, xs')$ to derive that $unfoldr\ uncons\ (x : xs') = x : unfoldr\ uncons\ xs'$. We conclude:

$$\begin{aligned}
(unfoldr\ uncons\ xs) \mathbin{!!} n &= (x : unfoldr\ uncons\ xs') \mathbin{!!} n \\
&= (unfoldr\ uncons\ xs') \mathbin{!!} (n - 1) && \text{(definition of (!!))} \\
&= xs' \mathbin{!!} (n - 1) && \text{(induction hypothesis)} \\
&= (x : xs') \mathbin{!!} n && \text{(definition of (!!))} \\
&= xs \mathbin{!!} n.
\end{aligned}$$

□