Part A (25 Marks)

Answer the following questions in a couple of short sentences. No need to be verbose.

1. (3 Marks) What is the difference between a *partial function* and *partial application*?

2. (3 Marks) Name *two* methods of measuring program coverage of a test suite, and explain how they differ.

3. (3 Marks) How are multi-argument functions typically modelled in Haskell?

4. (3 Marks) Is the type of $getChar$getChar below a pure function? Why or why not?

$$getChar :: \mathtt{IO\ Char}$$

getChar :: IO Char

5. (3 Marks) What is a *functional correctness* specification?

6. (3 Marks) Under what circumstances is performance important for an abstract model?

7. (3 Marks) What is the relevance of termination for the Curry-Howard correspondence?

8. (4 Marks) Imagine you are working on some price tracking software for some company stocks. You have already got a list of stocks to track pre-defined.

$$\mathbf{data}\ Stock = \mathsf{GOOG} \mid \mathsf{MSFT} \mid \mathsf{APPL}$$
$$stocks = [\mathsf{GOOG}, \mathsf{MSFT}, \mathsf{APPL}]$$

data Stock = GOOG | MSFT | APPL
stocks = [GOOG, MSFT, APPL]

Your software is required to produce regular reports of the stock prices of these companies. Your co-worker proposes modelling reports simply as a list of prices:

$$\mathbf{type}\ Report = [Price]$$

type Report = [Price]

Where each price in the list is the stock price of the company in the corresponding position of the *stocks* list. How is this approach potentially unsafe? What would be a safer representation?

Part B (25 Marks)

The following questions pertain to the given Haskell code:

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ z\ (x : xs) \quad = \quad f\ x\ (foldr\ f\ z\ xs) \quad \text{--} \quad (1) \\
&foldr\ f\ z\ [] \quad = \quad z \quad \text{--} \quad (2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathrm{foldr}\ f\ z\ (x : \mathrm{xs}) \quad = \quad f\ x\ (\mathrm{foldr}\ f\ z\ \mathrm{xs}) \quad \text{--} \quad (1) \\
&\mathrm{foldr}\ f\ z\ [] \quad = \quad z \quad \text{--} \quad (2)
\end{aligned}
$$

1. (3 Marks) State the type, if one exists, of the expression $foldr\ (:)\ ([] :: [\mathrm{Bool}])$ foldr (:) ([] :: [Bool]).

2. (4 Marks) Show the evaluation of $foldr\ (:)\ []\ [1, 2]$ foldr (:) [] [1, 2] via equational reasoning.

3. (2 Marks) In your own words, describe what the function $foldr\ (:)\ []$ foldr (:) [] does.

4. (12 Marks) We shall prove by induction on lists that, for all lists xs xs and ys ys:

$$foldr\ (:)\ xs\ ys = ys \mathbin{++} xs$$

$$\mathrm{foldr}\ (:)\ \mathrm{xs}\ \mathrm{ys} = \mathrm{ys} \mathbin{++} \mathrm{xs}$$

   i. (3 Marks) First show this for the base case where $ys = []$ ys = [] using equational reasoning. You may assume the left identity property for $++$ ++, that is, for all ls ls:

$$ls = [] \mathbin{++} ls$$

$$\mathrm{ls} = [] \mathbin{++} \mathrm{ls}$$

   ii. (9 Marks) Next, we have the case where $ys = (k : ks)$ ys = (k : ks) for some item k k and list ks ks.

      a. (3 Marks) What is the *inductive hypothesis* about ks ks?

b. (6 Marks) Using this inductive hypothesis, prove the above theorem for the inductive case using equational reasoning.

5. (2 Marks) What is the time complexity of the function call $foldr\ (:)\ []\ xs$ foldr (:) [] xs, where xs xs is of size n n?

6. (2 Marks) What is the time complexity of the function call $foldr\ (\lambda a\ as \rightarrow as \mathbin{+\!+} [a])\ []\ xs$ foldr (λa as → as ++ [a]) [] xs, where xs xs is of size n n

Part C (25 Marks)

A *sparse vector* is a vector where a lot of the values in the vector are zero. We represent a sparse vector as a list of position-value pairs, as well as an `Int` Int to represent the overall length of the vector:

$$\textbf{data}\ SVec = \mathsf{SV}\ \mathsf{Int}\ [(\mathsf{Int}, \mathsf{Float})]$$

data SVec = SV Int [(Int, Float)]

We can convert a sparse vector back into a dense representation with this $expand$ expand function:

$$\begin{array}{l}
expand :: SVec \rightarrow [\mathsf{Float}] \\
expand\ (\mathsf{SV}\ n\ vs) = map\ check\ [0\,..\,n-1] \\
\quad \textbf{where} \\
\qquad check\ x = \textbf{case}\ lookup\ x\ vs\ \textbf{of} \\
\qquad\qquad \begin{array}{lll} \mathsf{Nothing} & \rightarrow & 0 \\ \mathsf{Just}\ v & \rightarrow & v \end{array}
\end{array}$$

expand :: SVec → [Float]
expand (SV n vs) = map check [0 . . n − 1]
where
check x = case lookup x vs of
Nothing → 0
Just v → v

For example, the $SVec$ SVec value $\mathsf{SV}\ 5\ [(0, 2.1), (4, 10.2)]$ SV 5 [(0, 2.1), (4, 10.2)] is expanded into $[2.1, 0, 0, 0, 10.2]$

1. (16 Marks) There are a number of $SVec$ values that do not correspond to a meaningful vector - they are invalid.
   i. (6 Marks) Which two *data invariants* must be maintained to ensure validity of an $SVec$ value? Describe the invariants in informal English.

   ii. (4 Marks) Give two examples of $SVec$ values which violate these invariants.

   iii. (6 Marks) Define a Haskell function $wellformed :: SVec \rightarrow Bool$ which returns `True` iff the data invariants hold for the input $SVec value$. Your Haskell doesn't have to be syntactically perfect, so long as the intention is clear.
   You may find the function $nub :: (\text{Eq } a) \Rightarrow [a] \rightarrow [a]$ useful, which removes duplicates from a list.

2. (9 Marks) Here is a function to multiply a $SVec$ vector by a scalar:

$$vsm :: SVec \rightarrow Float \rightarrow SVec$$
$$vsm\ (\mathsf{SV}\ n\ vs)\ s = \mathsf{SV}\ n\ (map\ (\lambda(p, v) \rightarrow (p, v * s))\ vs)$$

$$\text{vsm} :: \text{SVec} \rightarrow \text{Float} \rightarrow \text{SVec}$$
$$\text{vsm}\ (\text{SV}\ n\ \text{vs})\ s = \text{SV}\ n\ (\text{map}\ (\lambda(p, v) \rightarrow (p, v * s))\ \text{vs})$$

   i. (3 Marks) Define a function $vsmA$ that performs the same operation, but for dense vectors (i.e. lists of Float).

   ii. (6 Marks) Write a set of properties to specify *functional correctness* of this function. *Hint:* All the other functions you need to define the properties have already been mentioned in this part. It should maintain data invariants as well as refinement from the abstract model.

Part D (25 Marks)

1. (10 Marks) Imagine you are working for a company that maintains this library for a database of personal records, about their customers, their staff, and their suppliers.

$$\begin{aligned}&\textbf{newtype}\ Person = \ldots\\[4pt] &name :: Person \rightarrow String\\ &salary :: Person \rightarrow \mathsf{Maybe}\ String\\ &fire :: Person \rightarrow \mathsf{IO}\ ()\\ &company :: Person \rightarrow \mathsf{Maybe}\ String\end{aligned}$$

newtype Person = …

name :: Person → String
salary :: Person → Maybe String
fire :: Person → IO ()
company :: Person → Maybe String

The $salary$ function returns Nothing if given a person who is not a member of company staff. The $fire$ function will also perform no-op unless the given person is a member of company staff. The $company$ function will return Nothing unless the given person is a supplier.
Rewrite the above type signatures to enforce the distinction between the different types of person statically, within Haskell's type system. The function $name$ must work with all kinds of people as input.
Hint: Attach *phantom* type parameters to the $Person$ type.

2. (15 Marks) Consider the following two types in Haskell:

$$\begin{array}{l}
\textbf{data}\ \mathit{List}\ a\ \textbf{where} \\
\quad \mathsf{Nil} \quad :: \quad \mathit{List}\ a \\
\quad \mathsf{Cons} \quad :: \quad a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ \mathit{Nat}\ =\ \mathsf{Z} \mid \mathsf{S}\ \mathit{Nat} \\
\textbf{data}\ \mathit{Vec}\ (n :: \mathit{Nat})\ a\ \textbf{where} \\
\quad \mathsf{VNil} \quad :: \quad \mathit{Vec}\ \mathsf{Z}\ a \\
\quad \mathsf{VCons} \quad :: \quad a \rightarrow \mathit{Vec}\ n\ a \rightarrow \mathit{Vec}\ (\mathsf{S}\ n)\ a
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ \mathrm{List}\ a\ \textbf{where} \\
\quad \mathrm{Nil} \quad :: \quad \mathrm{List}\ a \\
\quad \mathrm{Cons} \quad :: \quad a \longrightarrow \mathrm{List}\ a \longrightarrow \mathrm{List}\ a
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ \mathrm{Nat}\ =\ \mathrm{Z} \mid \mathrm{S}\ \mathrm{Nat} \\
\textbf{data}\ \mathrm{Vec}\ (n :: \mathrm{Nat})\ a\ \textbf{where} \\
\quad \mathrm{VNil} \quad :: \quad \mathrm{Vec}\ \mathrm{Z}\ a \\
\quad \mathrm{VCons} \quad :: \quad a \longrightarrow \mathrm{Vec}\ n\ a \longrightarrow \mathrm{Vec}\ (\mathrm{S}\ n)\ a
\end{array}$$

What is the difference between these types? In which circumstances would VecVec be the better choice, and in which $List$List?

i. (5 Marks)

ii. (5 Marks) Here is a simple list function:

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{zip} :: \mathit{List}\ a \rightarrow \mathit{List}\ b \rightarrow \mathit{List}\ (a, b)} \\
\mathit{zip}\ \ \mathsf{Nil} & \mathit{ys} & = & \mathsf{Nil} \\
\mathit{zip}\ \ \mathit{xs} & \mathsf{Nil} & = & \mathsf{Nil} \\
\mathit{zip}\ \ (\mathsf{Cons}\ x\ \mathit{xs}) & (\mathsf{Cons}\ y\ \mathit{ys}) & = & \mathsf{Cons}\ (x, y)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathrm{zip} :: \mathrm{List}\ a \longrightarrow \mathrm{List}\ b \longrightarrow \mathrm{List}\ (a, b)} \\
\mathrm{zip}\ \ \mathrm{Nil} & \mathrm{ys} & = & \mathrm{Nil} \\
\mathrm{zip}\ \ \mathrm{xs} & \mathrm{Nil} & = & \mathrm{Nil} \\
\mathrm{zip}\ \ (\mathrm{Cons}\ x\ \mathrm{xs}) & (\mathrm{Cons}\ y\ \mathrm{ys}) & = & \mathrm{Cons}\ (x, y)\ (\mathrm{zip}\ \mathrm{xs}\ \mathrm{ys})
\end{array}$$

Define a new version of zipzip which operates on VecVec instead of $List$List wherever possible. You can constrain the lengths of the input.

iii. (5 Marks) Here is another list function:

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathsf{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a \\
&\mathit{filter}\ p\ \mathsf{Nil} = \mathsf{Nil} \\
&\mathit{filter}\ p\ (\mathsf{Cons}\ x\ \mathit{xs}) \\
&\qquad \mid p\ x \quad = \quad \mathsf{Cons}\ x\ (\mathit{filter}\ p\ \mathit{xs}) \\
&\qquad \mid \mathit{otherwise} \quad = \quad \mathit{filter}\ p\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\text{filter} :: (a \to \text{Bool}) \to \text{List}\ a \to \text{List}\ a \\
&\text{filter}\ p\ \text{Nil} = \text{Nil} \\
&\text{filter}\ p\ (\text{Cons}\ x\ \text{xs}) \\
&\qquad \mid p\ x \quad = \quad \text{Cons}\ x\ (\text{filter}\ p\ \text{xs}) \\
&\qquad \mid \text{otherwise} \quad = \quad \text{filter}\ p\ \text{xs}
\end{aligned}
$$

Define a new version of filterfilter which operates on VecVec instead of ListList wherever possible.

Part E (25 Marks)

1. (10 Marks) An applicative functor is called *commutative* iff the order in which actions are sequenced does not matter. In addition to the normal applicative laws, a *commutative* applicative functor satisfies:

$$f \langle\$\rangle\ u\ \langle * \rangle\ v \;=\; \mathit{flip}\ f\ \langle\$\rangle\ v\ \langle * \rangle\ u$$

$$f \langle\$\rangle\ u\ \langle * \rangle\ v \;=\; \text{flip}\ f\ \langle\$\rangle\ v\ \langle * \rangle\ u$$

i. (2 Marks) Is the `Maybe`Maybe Applicative instance *commutative*? Explain your answer.

ii. (3 Marks) We have seen two different `Applicative`Applicative instances for lists. Which of these instances, if any, are *commutative*? Explain your answer.

iii. (5 Marks) A *commutative* monad is the same as a commutative applicative, only specialised to monads. Express the commutativity laws above in terms of monads, using either **do**do notation or the raw pure/bind functions.

2. (10 Marks) Translate the following logical formulae into types, and provide Haskell types that correspond to proofs of these formulae, if one exists. If not, explain why not.

i. (2 Marks) $(A \vee B) \rightarrow (B \vee A)$$(A \vee B) \rightarrow (B \vee A)$

ii. (2 Marks) $(A \vee A) \rightarrow A$$(A \vee A) \rightarrow A$

iii. (3 Marks) $(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))$
$(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))$

iv. (3 Marks) $\neg((A \rightarrow \bot) \vee A)$$\neg((A \rightarrow \bot) \vee A)$

3. (5 Marks) Here is a Haskell data type:

$$\begin{array}{lcl} \textbf{data } X & = & \text{First () A} \\ & | & \text{Second () Void} \\ & | & \text{Third (Either B ())} \end{array}$$

$$\begin{array}{lcl} \textbf{data } X & = & \text{First () A} \\ & | & \text{Second () Void} \\ & | & \text{Third (Either B ())} \end{array}$$

Using known type isomorphisms, simplify this type as much as possible.

END OF SAMPLE EXAM

(don't forget to save!)

Time Remaining

2h 9m 33s

Save