Student Number: ____________________

Full Name: ____________________

Signature: ____________________

THE UNIVERSITY OF NEW SOUTH WALES

COMP3141

Software System Design and Implementation

Sample Final Exam

Time Allowed: 2 hours, excluding reading time.

Reading Time: 10 minutes.

Total Number of Questions: 4

Answer **all** questions.

The questions **are** of equal value.

You are permitted **two** hand-written, single-sided A4 sheets of notes.

Only write your answers on the provided booklets.

Answers must be written in ink, with the exception of diagrams.

Drawing instruments or rules may be used.

Excessively verbose answers may lose marks.

There is a 3% penalty if your name and student number are not filled in correctly.

Papers and written notes may not be retained by students.

Question 1 (25 marks)

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

(a) (3 marks) What is the difference between a *partial function* and *partial application*?

> **Solution:** A partial function is a function not defined for its whole domain, whereas partial application is where a n-argument function is applied to fewer than n values.

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

> **Solution:** Function coverage measures whether or not a given function is executed by the test suite, whereas path coverage measures all possible execution paths. Function coverage is easier to compute than path coverage.

(c) (3 marks) How are multi-argument functions typically modelled in Haskell?

> **Solution:** Mutliple argument functions are usually modelled by one-argument functions that in turn return functions to accept further arguments. For example, the add function `(+) :: Int -> (Int -> Int)` takes an `Int` and returns a function.

(d) (3 marks) Is the type of *getChar* below a pure function? Why or why not?

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

> **Solut**
> not a function. It is instead an `IO` proced

(e) (3 marks) What is a *functional correctness*

> **Solution:** A functional correctness specification is a set of properties that completely specify the definition of correctness for a program. It is often expressed as the combination of data invariants and a refinement from an abstract model.

(f) (3 marks) Under what circumstances is performance important for an abstract model?

> **Solution:** If we are testing using an abstract model (for example with QuickCheck), then we are still computing with it and thus intractable or uncomputable abstract models would not be suitable.

(g) (3 marks) What is the relevance of termination for the Curry-Howard correspondence?

> **Solution:** The Curry-Howard correspondence assumes that function types are pure and total. Non-termination makes the logic the type system corresponds to inconsistent.

(h) (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.

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

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:

$$\textbf{type}\ \mathit{Report} = [\mathit{Price}]$$

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?

Solution: It is not guaranteed that the prices will line up to the stocks, or that there will be a stock for every price and a price for every stock. An alternative would be:

$$\textbf{type}\ \mathit{Report} = [(\mathit{Stock}, \mathit{Price})]$$

which at least ensures that the right stocks are associated with the right price.

Question 2 (25 marks)

The following questions pertain to the given Haskell code:

$$\begin{array}{lcll} \mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b & & & \\ \mathit{foldr}\ f\ z\ (x : xs) & = & f\ x\ (\mathit{foldr}\ f\ z\ xs) & \texttt{--}\ \ (1) \\ \mathit{foldr}\ f\ z\ [] & = & z & \texttt{--}\ \ (2) \end{array}$$

(a) (3 marks) State the type, if one exists, of the expression $\mathit{foldr}\ (:)\ ([] :: [\mathsf{Bool}])$.

Solut

(b) (4 marks) Sho oning.

Solution:

$$\begin{array}{cll} & \mathit{foldr}\ (:)\ []\ [& \\ = & 1 : (\mathit{foldr}\ (& \\ = & 1 : 2 : (\mathit{foldr}\ (:)\ []\ []) & (1) \\ = & 1 : 2 : [] & (2) \end{array}$$

(c) (2 marks) In your own words, describe what the function $\mathit{foldr}\ (:)\ []$ does.

Solution: It is the identity function for lists.

(d) (12 marks) We shall prove by induction on lists that, for all lists xs and ys:

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

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

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

Solution:

$$\begin{array}{rcll} \mathit{foldr}\ (:)\ xs\ [] & = & xs & (2) \\ & = & [] \mathbin{+\!\!+} xs & (\mathsf{id}) \end{array}$$

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

α) (3 marks) What is the *inductive hypothesis* about ks?

Solution:

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

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

Solution:

$$\begin{array}{llr} foldr\ (:)\ xs\ (k : ks) & = (:)\ k\ (foldr\ (:)\ xs\ ks) & (1) \\ & = k : (foldr\ (:)\ xs\ ks) & \text{simp} \\ & = k : (ks \mathbin{+\!\!+} xs) & \text{I.H} \\ & = (k : ks) \mathbin{+\!\!+} xs & \text{def. of } (\mathbin{+\!\!+}) \end{array}$$

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

Solution: $\mathcal{O}(n)$

(f) (2 marks) What is the time complexity of the function call $foldr\ (\lambda a\ as \to as \mathbin{+\!\!+} [a])\ []\ xs$, where xs

Solut

Question 3 (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` to represent the overall length of the vector:

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

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

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

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

(a) (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.

> **Solution:** For a value $\mathsf{SV}\ n\ vs$, the list vs should contain only one value for a given position, and the positions of all values in vs should all be ≥ 0 and $< n$.

ii. (4 mark variants.

> S 1.1), (1, 5.3)]
> Vi

iii. (6 marks) Define a Haskell function *we* hich returns `True` iff the data invariants hold for the input esn't have to be syntactically perfect, so long as the intention is cl $nub :: (\mathsf{Eq}\ a) \Rightarrow [a] \to [a]$ useful, which removes duplicates from a list.

> **Solution:**
>
> $$\begin{array}{l}
wellformed\ (\mathsf{SV}\ n\ vs) \\
\quad = \textbf{let}\ ps = map\ fst\ vs \\
\qquad \textbf{in}\ all\ (\lambda x \to x \geq 0\ \texttt{\&\&}\ x < n)\ ps \\
\qquad \texttt{\&\&}\ nub\ ps\ \texttt{==}\ ps
\end{array}$$

(b) (9 marks) Here is a function to multiply a $SVec$ vector by a scalar:

$$\begin{array}{l}
vsm :: SVec \to Float \to SVec \\
vsm\ (\mathsf{SV}\ n\ vs)\ s = \mathsf{SV}\ n\ (map\ (\lambda(p, v) \to (p, v * s))\ vs)
\end{array}$$

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

> **Solution:**
>
> $$vsmA\ xs\ s = map\ (*\ s)\ xs$$

ii. (6 marks) Write a set of properties to specify *functional correctness* of the function vsm.
Hint: All the other functions you need to define the properties have already been

mentioned on this page. It should maintain data invariants as well as refinement from the abstract model of dense vectors.

Solution: Data invariants:

$$\mathit{wellformed}\ xs \implies \mathit{wellformed}\ (\mathit{vsm}\ xs\ s)$$

Data refinement:

$$\mathit{expand}\ (\mathit{vsm}\ xs\ s) == \mathit{vsmA}\ (\mathit{expand}\ xs)\ s$$

Question 4 (25 marks)

(a) (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.

$$\textbf{newtype}\ \mathit{Person} = \ldots$$

$$\begin{aligned}&\mathit{name} :: \mathit{Person} \to \mathit{String}\\ &\mathit{salary} :: \mathit{Person} \to \mathsf{Maybe}\ \mathit{String}\\ &\mathit{fire} :: \mathit{Person} \to \mathsf{IO}\ ()\\ &\mathit{company} :: \mathit{Person} \to \mathsf{Maybe}\ \mathit{String}\end{aligned}$$

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.

Solution:

$$\begin{aligned}&\textbf{data}\ \mathit{Staff}\\ &\textbf{data}\ \mathit{Customer}\\ &\textbf{data}\ \mathit{Supplier}\\ &\textbf{newtype}\ \mathit{Person}\ t = \ldots\end{aligned}$$

-- Maybe no longer n

$\mathit{company} :: \mathit{Person}\ S$

(b) (15 marks) Consider the following two types in Haskell:

$$\begin{aligned}&\textbf{data}\ \mathit{List}\ a\ \textbf{where}\\ &\quad \mathsf{Nil} \quad :: \mathit{List}\ a\\ &\quad \mathsf{Cons} :: a \to \mathit{List}\ a \to \mathit{List}\ a\end{aligned}$$

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

i. (5 marks) What is the difference between these types? In which circumstances would *Vec* be the better choice, and in which *List*?

Solution: `Vec` tracks its length in the type level, whereas `List` does not. Usually, one would only use `Vec` if one frequently needed to express static preconditions about the length of the list. This way, the type checker can ensure these preconditions are met automatically.

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

$$\begin{aligned}&\mathit{zip} :: \mathit{List}\ a \to \mathit{List}\ b \to \mathit{List}\ (a, b)\\ &\mathit{zip}\ \ \mathsf{Nil}\ \ \ \mathit{ys} = \mathsf{Nil}\\ &\mathit{zip}\ \ \mathit{xs}\ \ \ \mathsf{Nil} = \mathsf{Nil}\\ &\mathit{zip}\ \ (\mathsf{Cons}\ x\ \mathit{xs})\ \ (\mathsf{Cons}\ y\ \mathit{ys}) = \mathsf{Cons}\ (x, y)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})\end{aligned}$$

Define a new version of *zip* which operates on *Vec* instead of *List* wherever possible. You can constrain the lengths of the input.

Solution:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{zipV} :: \mathit{Vec}\ n\ a \to \mathit{Vec}\ n\ b \to \mathit{Vec}\ n\ (a, b)} \\
\mathit{zipV}\ \ \mathsf{VNil} & \mathit{ys} & = & \mathsf{VNil} \\
\mathit{zipV}\ \ \mathit{xs} & \mathsf{VNil} & = & \mathsf{VNil} \\
\mathit{zipV}\ \ (\mathsf{VCons}\ x\ \mathit{xs}) & (\mathsf{VCons}\ y\ \mathit{ys}) & = & \mathsf{VCons}\ (x, y)\ (\mathit{zipV}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{filter} :: (a \to \mathsf{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a} \\
\multicolumn{3}{l}{\mathit{filter}\ p\ \mathsf{Nil} = \mathsf{Nil}} \\
\multicolumn{3}{l}{\mathit{filter}\ p\ (\mathsf{Cons}\ x\ \mathit{xs})} \\
\quad \mid p\ x & = & \mathsf{Cons}\ x\ (\mathit{filter}\ p\ \mathit{xs}) \\
\quad \mid \mathit{otherwise} & = & \mathit{filter}\ p\ \mathit{xs}
\end{array}
$$

Define a new version of *filter* which operates on *Vec* instead of *List* wherever possible.

Solution: We can't return a Vec because we don't know how long it will be.

$$\mathit{filterV} :: (a \quad \mathsf{Bool}) \quad \mathit{Vec}\ n\ a \quad \mathit{List}\ a$$

s)

END OF EXA